Bases formelles du TAL

Pierre-Léo Bégay
pbegay@ens-cachan.fr

12 mai 2020

La théorie des automates est l'algèbre linéaire de l'informatique [...], connaissance de base, fondamentale, connue de tous et utilisée par tous, qui fait partie du paysage intellectuel depuis si longtemps qu'on ne l'y remarquerait plus.

Jacques Sakarovitch, dans l'avant-propos de Eléments de théorie des automates

"Peux-tu expliquer correctement la différence entre le présent et le passé du conditionnel en anglais ? me demanda-t-elle soudain.
- Je crois que oui, lui répondis-je.
- Je voudrais bien savoir à quoi cela te sert dans ta vie quotidienne.
- C'est vrai que cela ne m'est pas très utile, mais je crois que, plutôt que d'en chercher l'utilité concrète, il faut le considérer comme un exercice destiné à faire vous faire appréhender les choses avec méthode"

Haruki Murakami, La ballade de l'impossible (traduit du japonais par Rose-Marie Makino-Fayolle)

Table des matières

1 Langages **4**
- 1.1 Mots . . . 4
- 1.2 Langage . . . 8

2 Expressions régulières **10**
- 2.1 Lexique et idée générale . . . 10
  - 2.1.1 Les lettres et ϵ, la base . . . 10
  - 2.1.2 ., la concaténation . . . 10
  - 2.1.3 *, l'itération . . . 11
  - 2.1.4 +, la disjonction . . . 12
- 2.2 Syntaxe . . . 13
- 2.3 Sémantique . . . 15
  - 2.3.1 Les cas de base . . . 15
  - 2.3.2 Sémantique de la concaténation . . . 16
  - 2.3.3 Sémantique de la disjonction . . . 16
  - 2.3.4 Sémantique de l'itération . . . 17
- 2.4 Mise en application . . . 17
  - 2.4.1 Quelques astuces . . . 18
  - 2.4.2 Syntaxe en pratique . . . 18
  - 2.4.3 Calcul de l'appartenance . . . 19

3 Automates finis **20**
- 3.1 Automates finis déterministes . . . 20
  - 3.1.1 Principe général . . . 20
  - 3.1.2 Formalisation et implémentation . . . 23
- 3.2 Automates finis non-déterministes . . . 25
  - 3.2.1 Principe général . . . 25
  - 3.2.2 Formalisation et implémentation . . . 27
- 3.3 Transformation d'automates . . . 28
  - 3.3.1 Complétion . . . 29
  - 3.3.2 Déterminisation . . . 29
  - 3.3.3 Minimisation . . . 31
- 3.4 Limite de la reconnaissance par automates finis . . . 34
- 3.5 Propriétés de clôture des langages reconnus par automates . . . 36
  - 3.5.1 Union . . . 36
  - 3.5.2 Concaténation . . . 36
  - 3.5.3 Itération . . . 38

3.5.4 Intersection . . . 39
3.5.5 Complémentaire . . . 41
3.5.6 Conséquences pour les langages non-reconnus . . . 42

4 Théorème de Kleene **44**
4.1 Des expressions rationnelles aux automates . . . 44
4.1.1 Traduction récursive . . . 44
4.1.2 Traduction linéaire . . . 45
4.1.3 Méthode des résiduels . . . 49
4.2 Des automates aux expressions rationnelles - algorithme de McNaughton et Yamada 55

5 Grammaires formelles et hiérarchie de Chomsky **59**
5.1 Principe général . . . 59
5.2 Formalisation . . . 62
5.3 Hierarchie de Chomsky . . . 64
5.3.1 Grammaires de type 3, ou régulières . . . 64
5.3.2 Grammaires de type 2, ou non contextuelles . . . 67
5.3.3 Grammaires de type 1, ou contextuelles . . . 68
5.3.4 Grammaires de type 0, ou générales . . . 70
5.4 Utilisation de grammaires algébriques . . . 70
5.4.1 Arbres de dérivation . . . 70
5.4.2 Transformation de grammaires algébriques . . . 73

6 Notion(s) de calculabilité **80**
6.1 Différents modèles de calcul . . . 80
6.2 Un peu d'Histoire : le programme de Hilbert . . . 81
6.2.1 Complétude et cohérence . . . 81
6.2.2 Décidabilité et thèse de Church . . . 82
6.3 Et les automates alors ? . . . 84
6.3.1 Type 3 et automates finis . . . 84
6.3.2 Type 2 et automates à pile . . . 84
6.3.3 Type 1 et automates linéairement bornés . . . 85
6.3.4 Type 0 et machines de Turing . . . 85

A Rappels mathématiques **87**
A.1 Logique . . . 87
A.1.1 Raisonnement par l'absurde . . . 87
A.2 Ensembles . . . 88
A.2.1 Opérations entre ensembles . . . 88
A.2.2 Ensemble des parties . . . 89
A.3 Algorithmique . . . 90
A.3.1 Itératif vs. récursif, le cas des parcours d'arbre . . . 90

Chapitre 1

Langages

On définit d'abord la notion de mot, nécessaire à celle de langage. On verra ensuite comment décrire des langages à l'aide de notations ensemblistes, révisant ces dernières par la même occasion.

1.1 Mots

Définition 1.1.1: Mot

Un **mot** est une suite de lettres tirées d'un alphabet donné. L'ensemble des mots sur un alphabet Σ est noté Σ^*.

Exemple 1.1.1. *Etant donné l'alphabet $\Sigma = \{a, b, c\}$, on peut construire une infinité de mots parmi lesquels*

- *abc*
- *aab*
- *cc*
- *abcabcacbacbacbabcabcabcabcabcabcabc*
- *a*

Remarque On va s'intéresser ici à des langages et mots complètement abstraits, en général composés uniquement de a, b et c.

Définition 1.1.2: Mot vide

Une suite de lettres peut être de longueur zéro, formant alors **le mot vide**. Quel que soit l'alphabet, ce dernier sera noté ϵ.

Définition 1.1.3: Concaténation

L'opération de **concaténation**, notée ., consiste tout simplement à "coller" deux mots.

Exemple 1.1.2. *Quelques concaténations :*

— $ab.c = abc$
— $ab.ba = abba$

De plus, pour tout mot w,

$$w.\epsilon = \epsilon.w = w$$

Remarque Les algébristes enthousiastes remarqueront que $(\Sigma^*, ., \epsilon)$ forme un monoïde libre de base Σ

Définition 1.1.4: Longueur d'un mot

Etant donné un mot w, on note sa **longueur** $|w|$.

Exemple 1.1.3. *Tout naturellement,*
— $|abc| = 3$
— $|abba| = 4$
— $|c| = 1$
— $|\epsilon| = 0$

Définition 1.1.5: Principe d'induction sur un mot

Etant donnée une propriété P sur les mots. Si on a

1. $P(\epsilon)$ (cad. que P est vraie pour le mot vide)
2. $\forall w, \forall c \in \Sigma, (P(w) \rightarrow P(c.w))$ (cad. que si P est vraie pour un mot, alors elle reste vraie si on rajoute n'importe quelle lettre à gauche du mot)

Alors la propriété P est vraie pour tout mot w.

Remarque Est également valide le principe d'induction où, dans le cas récursif, la lettre est rajoutée à droite du mot plutôt qu'à sa gauche.

On va s'entraîner à utiliser ce principe d'induction en prouvant deux lemmes qui n'en nécessitaient sans doute pas tant :

Lemme 1. $\forall w \in \Sigma^*, |w| \geq 0$, *cad. que tout mot a une longueur positive.*

Démonstration. On procède par induction sur w.

Dans le cas de base, $w = \epsilon$. On a donc $|w| = |\epsilon| = 0 \geq 0$.

Dans le cas récursif, $w = c.w'$ avec $c \in \Sigma$ et on suppose $|w'| \geq 0$. On a $|c.w'| = 1 + |w'| \geq |w'| \geq 0$. □

Lemme 2. *Etant donnés deux mots w_1 et w_2, $|w_1.w_2| = |w_1| + |w_2|$.*

Démonstration. On procède par induction sur w_1.

Dans le cas de base, $w_1 = \epsilon$. On a donc $|w_1.w_2| = |\epsilon.w_2| = |w_2| = 0 + |w_2| = |w_1| + |w_2|$.

Dans le cas récursif, $w_1 = c.w_1'$ avec $c \in \Sigma$ et on suppose $|w_1'.w_2| = |w_1'| + |w_2|$. On a

	$\|w_1.w_2\|$	
=	$\|c.w_1'.w_2\|$	par définition de w_1
=	$1 + \|w_1'.w_2\|$	par définition de $\|.\|$
=	$1 + (\|w_1'\| + \|w_2\|)$	par hypothèse d'induction
=	$(1 + \|w_1'\|) + \|w_2\|$	par associativité de l'addition
=	$\|c.w_1'\| + \|w_2\|$	par définition de $\|.\|$
=	$\|w_1\| + \|w_2\|$	par définition de w_1

On a donc bien nos deux conditions pour le raisonnement par induction. □

Définition 1.1.6: Nombre d'occurrences d'une lettre

Etant donné un mot w et une lettre a, on note $|\mathbf{w}|_{\mathbf{a}}$ le nombre de a dans w.

Exemple 1.1.4. *On a*
- $|abc|_a = 1$
- $|abba|_b = 2$
- $|c|_a = 0$
- $|\epsilon|_a = 0$

Définition 1.1.7: Préfixe

Un mot p est un **préfixe** du mot w ssi $\exists v, w = p.v$, cad. ssi w *commence* par p.

Définition 1.1.8: Suffixe

Un mot s est un **suffixe** du mot w ssi $\exists v, w = v.s$, cad. ssi w *finit* par s.

Exemple 1.1.5. *Le mot abba admet comme préfixes ϵ, a, ab, abb et abba. Ses suffixes sont, quant à eux, ϵ, a, ba, bba et abba.*

Lemme 3. *$\forall w \in \Sigma^*, \epsilon$ et w sont des préfixes de w*

Démonstration. Pour ϵ, il suffit de prendre $v = w$. A l'inverse, en prenant $v = \epsilon$, on voit que w est son propre préfixe. □

Lemme 4. *$\forall w \in \Sigma^*, \epsilon$ et w sont des suffixes de w*

Démonstration. Analogue au lemme précédent. □

Exercice 1.1.1. *Combien de préfixes et suffixes admet un mot w quelconque ?*

Définition 1.1.9: Facteur

Un mot f est un **facteur** du mot w ssi $\exists v_1 v_2, w = v_1.f.v_2$, cad. ssi f apparaît dans w.

Exemple 1.1.6. *Les facteurs du mot abba sont ϵ, a, b, ab, ba, abb, bba et $abba$.*

Lemme 5. *$\forall w \in \Sigma^*$, ϵ et w sont des facteurs de w.*

Démonstration. Pour ϵ, il suffit de prendre $v_1 = w$ et $v_2 = \epsilon$ (ou l'inverse) et la condition est trivialement vérifiée. Pour w, on prend $v_1 = v_2 = \epsilon$. □

Exercice 1.1.2. *Donner l'ensemble des facteurs du mot abbba.*

Exercice 1.1.3. *(*) Donner la borne la plus basse possible du nombre de facteurs d'un mot w. Donner un mot d'au moins 3 lettres dont le nombre de facteurs est exactement la borne donnée.*

Définition 1.1.10: Sous-mot

Un mot s est un **sous-mot** du mot w ssi $w = v_0 s_0 v_1 s_1 v_2 ... s_n v_n$ et $s = s_0 s_1 ... s_n$, cad. ssi w est "s avec (potentiellement) des lettres en plus".

Exemple 1.1.7. *On souligne les lettres originellement présentes dans le sous-mot :*
- *ab est un sous-mot de baab, qu'on pourrait aussi voir comme baab*
- *abba est un sous-mot de baabaabba.*
- *ba n'est pas un sous-mot de aaabbb (l'ordre du sous-mot doit être préservé dans le mot)*

Lemme 6. *$\forall w \in \Sigma^*$, ϵ et w sont des sous-mots de w.*

Démonstration. Pour ϵ, il suffit de prendre $n = 0$, $s_0 = \epsilon$, $v_0 = w$ et $v_1 = \epsilon$ (ou l'inverse) et la condition est trivialement vérifiée. Pour w, on prend $n = 0$, $s_0 = w$ et $v_0 = v_1 = \epsilon$. □

Exercice 1.1.4. *Montrer que tout facteur d'un mot en est également un sous-mot. A l'inverse, montrer qu'un sous-mot n'est pas forcément un facteur.*

Exercice 1.1.5. *Donner toutes les façons de voir abba comme sous-mot de baaabaabbaa (cf. exemple 1.1.7).*

Exercice 1.1.6. *Donner l'ensemble des sous-mots de abba*

Exercice 1.1.7. *(*) Donner la borne la plus basse possible du nombre de sous-mots d'un mot w. Donner un mot dont le nombre de sous-mots est exactement la borne donnée.*

Exercice 1.1.8. *(*) Dans l'exercice 1.1.1, on demande le nombre exact de préfixes et suffixes d'un mot, alors que dans les exercices 1.1.3 et 1.1.7, on demande une borne, pourquoi ?*

1.2 Langage

Définition 1.2.1: Langage

Un langage, c'est un ensemble de mots.

On distingue donc d'entrée les deux langages extrêmes : $\mathbf{\Sigma}^*$, l'ensemble (infini) de tous les mots formés à partir de Σ, et $\emptyset$, le langage / ensemble vide, qui se caractèrise comme ne contenant aucun élément.

Remarque Ne surtout pas confondre $\emptyset$ et $\{\epsilon\}$. Le premier est un ensemble vide, contenant donc **0** élément, trandis que le second contient **1** élément, le mot ide.

Définition 1.2.2: Produit de langages

Le produit de deux langages L_1 et L_2, noté $L_1.L_2$, renvoie l'ensemble des mots composés d'un mot de L_1 puis d'un de L_2 :
$L_1.L_2 = \{w_1.w_2 \mid w_1 \in L_1 \ \wedge \ w_2 \in L_2\}$
Il s'agit d'un cas particulier de produit d'ensembles (cf. définition A.2.1.

Exemple 1.2.1. *Soit $L_1 = \{ab, b, \epsilon\}$ et $L_2 = \{a, b, aa\}$, on a*

$$\begin{array}{ll} & L_1.L_2 \\ = & \{ab.a, ab.b, ab.aa, b.a, b.b, b.aa, \epsilon.a, \epsilon.b, \epsilon.aa\} \\ = & \{aba, abb, abaa, ba, bb, baa, a, b, aa\} \end{array}$$

Le produit de langage peut être itéré [1] :

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^{n+1} &= L^n.L \end{aligned}$$

Les langages disposent en plus d'un opérateur spécial :

Définition 1.2.3: Etoile de Kleene

Soit L un langage. On note L^* la concaténation de n'importe quel nombre de mots apparaissant dans L, cad.

$$L^* = \bigcup_{n \in \mathbf{N}} L^n = L^0 \cup L^1 \cup L^2 \cup ...$$

1. Concrètement, les puissances sur les langages ont le même sens que sur les nombres, avec la multiplication remplacée par la concaténation

Exemple 1.2.2. *Soit $L = \{aa, b\}$, on a*

$$\begin{aligned} L^* = & \{epsilon\} \\ \cup & \{aa, b\} \\ \cup & \{aaaa, aab, baa, bb\} \\ \cup & \{aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbaa, bbb\} \\ \cup & \ldots \end{aligned}$$

La question maintenant est maintenant de savoir comment on définit et parle de langages précis et plus "intermédiaires" que les deux précédents. En tout généralité, les ensembles peuvent être définis de façon **extensionnelle** ou **intentionnelle**.

Définition 1.2.4: Définition extensionnelle d'un ensemble

On **définit extensionnellement un ensemble** en en donnant la liste des éléments. L'ensemble vide se note quant à lui $\emptyset$.

Exemple 1.2.3. *On définit par exemple l'ensemble (sans intérêt) suivant :*

$$A = \{b, aca, abba\}$$

Les définitions extensionnelles ont le mérite d'être pour le moins simples, mais pas super pratiques quand il s'agit de définir des ensembles avec un nombre infini d'éléments, comme l'ensemble des mots de longueur pair.

Définition 1.2.5: Définition intensionnelle d'un ensemble

On **définit intensionnellement un ensemble** à l'aide d'une propriété que tous ses éléments satisfont. Étant donnés une propriété $Q(x)$ (typiquement représentée sous la forme d'une formule logique) et un ensemble A, on note $\{x \in A \mid Q(x)\}$ l'ensemble des éléments de A qui satisfont P. Si l'ensemble A est évident dans le contexte, on s'abstiendera de le préciser.

Exemple 1.2.4. *On peut définir l'ensemble des mots de longueur paire $\{w \in \Sigma^* \mid |w| \text{ pair}\}$.*

Si les définitions intentionnelles permettent, contrairement aux extensionnelles, de dénoter des ensembles contenant une infinité de mots, elles sont avant tout un outil théorique. En effet, une propriété comme "$|w|$ paire" ne dit rien à un ordinateur en soi, et doit donc être définie formellement. Se pose alors la question d'un langage pour les propriétés.

Plusieurs logiques équipées des bonnes primitives peuvent être utilisées, mais les traductions sont rarement très agréables. Certaines propriétés nécessitent en effet de ruser contre le langage, voire sont impossibles à formaliser dans certaines logiques. Il existe heureusement un outil qui va nous aider, avec le premier problème du moins.

Chapitre 2

Expressions régulières

Les expressions régulières permettent définir de façon finie - et relativement intuitive - "la forme" des mots d'un langage, potentiellement infini. On en présentera d'abord le lexique et l'idée générale à l'aide d'exemples, puis on en définira formellement la syntaxe et la sémantique.

2.1 Lexique et idée générale

Une expression rationnelle (ou *regex*, pour *regular expression*[1]) est, en gros, une *forme de mot*, écrite à l'aide de lettres et des symboles ., $*$ et $+$.

2.1.1 Les lettres et ϵ, la base

Les regex sont construites récursivement, en partant bien sûr des cas de base. Étant donné un alphabet Σ, ces derniers sont les différentes lettres de Σ, ainsi que ϵ. Ces regex dénotent chacune un seul mot, la lettre utilisée dans le premier cas, et le mot vide dans le second.

Exemple 2.1.1. *La regex a dénote le langage* $\{a\}$.

2.1.2 ., la concaténation

On peut heureusement concaténer des regex en utilisant à nouveau le symbole .. La concaténation de deux expressions rationnelles e_1 et e_2, notée $e_1.e_2$ donc, dénote l'ensemble des mots qui peuvent se décomposer en une première partie "de e_1" et une deuxième "de e_2".

Remarque En pratique, on ne notera pas les . dans les regex, mais quelque chose comme *abbc* devrait en théorie être lu comme *a.b.b.c*

Exemple 2.1.2. *La regex abca dénote l'ensemble* $\{abca\}$.

1. On se trompera sans doute souvent en parlant d'"expression régulière"

2.1.3 *, l'itération

Le symbole $*$ permet de dire qu'une regex peut être répétée autant de fois que voulu (y compris 0).

Exemple 2.1.3. *La regex ab^*c dénote l'ensemble des mots de la forme "un a, puis une série (éventuellement vide) de b, puis un c", c'est-à-dire $\{ac, abc, abbc, abbbc, ...\}$*

En utilisant des parenthèses, on peut appliquer $*$ à des facteurs entiers :

Exemple 2.1.4. *La regex $(aa)^*$ dénote l'ensemble des mots de la forme "une série (éventuellement vide) de deux a", c'est-à-dire $\{\epsilon, aa, aaaa, aaaaaa, ...\}$, ou encore les mots composés uniquement de a et de longueur paire.*

On peut bien sûr utiliser plusieurs $*$ dans une même expression. Dans ce cas, les nombres de "copies" des facteurs concernés ne sont pas liés, comme l'illustrent les examples suivant :

Exemple 2.1.5. *La regex a^*b^* dénote l'ensemble des mots de la forme "Une série (éventuellement vide) de a, puis une série (éventuellement vide) de b", contenant notamment ϵ, a, b, aaab, abbbb, bb, aaa, ab, aabb et aaabb*

Exemple 2.1.6. *La regex $ab(bab)^*b(ca)^*b$ dénote l'ensemble des mots de la forme "ab, puis une série (éventuellement vide) de bab, puis un b, puis une série (éventuellement vide) de ca, puis un b", contenant notamment abbb, abbabbb, abbcab ou abbabbabbabbcab.*

Exercice 2.1.1. *Donner 5 autres mots appartenant au langage dénotée par l'expression de l'exemple 2.1.6.*

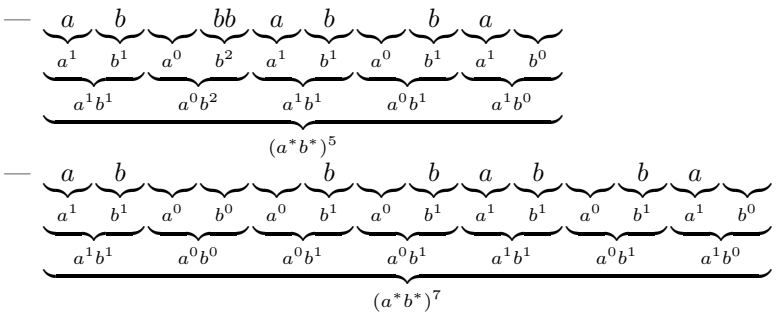
Exercice 2.1.2. *Pourquoi le changement de formulation dans les exemples 2.1.5 et 2.1.6 par rapport aux exemples précédents ("c'est à dire $\{x, y, z...\}$" qui devient "contenant notamment x, y ou z") ?*

On peut même faire encore plus rigolo, en enchâssant les étoiles :

Exemple 2.1.7. *La regex $(a^*b^*)^*$ dénote l'ensemble des mots de la forme "une série (éventuellement vide) de [(une série (éventuellement vide) de a] puis [une série (éventuellement vide) de b]]", contenant notamment ϵ, abba ; baba ou abbbabba.*

Remarque Certaines regex peuvent générer des mêmes mots de plusieurs façons. Si on prend l'expression de l'exemple 2.1.7 et le mot *abbbabba*, on peut la voir comme

$$\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{bbb}_{b^3}}_{a^1b^3}\underbrace{\underbrace{a}_{a^1}\underbrace{bb}_{b^2}}_{a^1b^2}\underbrace{\underbrace{a}_{a^1}\underbrace{}_{b^0}}_{a^1b^0}}_{(a^*b^*)^3}$$



Le premier déroulement (on parlera de **dérivation**) semble bien sûr plus "naturel" et optimal que les deux autres. Ils sont pourtant tout aussi valides, et il sera utile en pratique d'éviter des expressions fortement ambiguës comme celle-ci.

2.1.4 +, la disjonction

Le symbole + permet quant à lui de signifier qu'on a le choix entre plusieurs sous-regex.

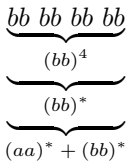
Exemple 2.1.8. *La regex $aa + bb$ dénote l'ensemble $\{aa, bb\}$*

Si on a par exemple $\Sigma = \{a, b, c\}$, alors $(a + b + c)$ correspond à "n'importe quelle lettre de l'alphabet". On écrira cette expression plus simplement Σ.

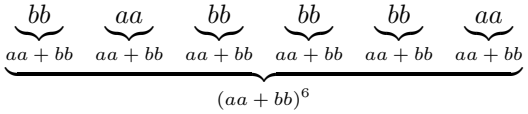
Exemple 2.1.9. *L'expression $\Sigma\Sigma\Sigma$, ou Σ^3, correspond à n'importe quel mot de longueur 3.*

La disjonction peut bien sûr se combiner avec $*$:

Exemple 2.1.10. *La regex $(aa)^* + (bb)^*$ dénote l'ensemble des mots composés uniquement de a et de longueur paire, ou uniquement de b et de longueur également paire, par exemple ϵ, aa, bb, aaaa ou bbbbbb. La dérivation du dernier mot serait alors de la forme*



Exemple 2.1.11. *La regex $(aa + bb)^*$ dénote l'ensemble des mots de la forme "une série (éventuellement vide) de aa et bb", par exemple aabb, aaaaaaaaaaaa ou bbaabbbbbbaa. La dérivation du dernier mot ressemblerait alors à*



Exercice 2.1.3. *Donner un mot acceptant deux dérivations avec la regex de l'exemple 2.1.10 (justifier en donnant les dérivations). Existe-t-il un autre mot admettant plusieurs dérivations ?*

Exercice 2.1.4. *Existe-t-il un mot acceptant plusieurs dérivations pour la regex de l'exemple 2.1.11 ?*

Exercice 2.1.5. *Donner un mot accepté par la regex de l'exemple 2.1.11 mais pas celle de l'exemple 2.1.10. Est-il possible de trouver un mot qui, à l'inverse, est accepté par la deuxième mais pas la première ?*

Exercice 2.1.6. *(*) Exprimer, en langue naturelle et de façon concise, le langage dénoté par la regex de l'exemple 2.1.7. Traduire ensuite ce langage en une regex non-ambiguë, c'est-à-dire où il n'y aura qu'une dérivation pour chaque mot.*

Notez que dans l'exemple 2.1.10, on choisit d'abord de composer le mot de a ou de b, puis la longueur. A l'inverse, on choisit dans la regex de l'exemple 2.1.11 la longueur, puis a ou b pour chaque "morceau", et ce, individuellement. Pour mieux comprendre cette différence, il faut s'intéresser formellement à la mécanique des regex, qui se décompose bien évidemment entre syntaxe et sémantique.

2.2 Syntaxe

Les expressions rationnelles ont, comme à peu près tout langage, une structure. Elles sont définies à l'aide de 5 règles, dont 3 récursives, qui correspondent au lexique décrit précédemment :

$$\begin{array}{rcl} e & ::= & \epsilon \\ & | & a \in \Sigma \\ & | & e_1.e_2 \\ & | & e_1 + e_2 \\ & | & e_1^* \end{array}$$

FIGURE 2.1 – Syntaxe des expression régulières

La figure 2.1 se lit "une expression rationnelle e est

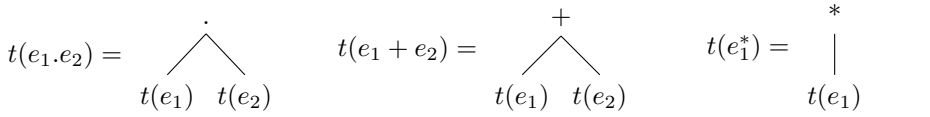
soit le symbole ϵ
soit une lettre appartenant à l'alphabet Σ
soit une expression rationnelle e_1 (définie à l'aide des mêmes règles), suivie de ., puis d'une expression rationnelle e_2
soit une expression rationnelle e_1, suivie de $+$, puis d'une expression rationnelle e_2
soit une expression rationnelle e_1 auréolée d'un $*$

et rien d'autre".

Ces règles de dérivation nous permettent de *parser* des expressions rationnelles. En notant $t()$ la fonction qui prend une regex et renvoie son arbre syntaxique, on peut la définir à l'aide des 5 règles de la figure 2.1 :

— $t(\epsilon)$ renvoie une feuille annotée par ϵ.
— $t(a)$ renvoie une feuille annotée par a.
— $t(e_1.e_2)$ renvoie un noeud . dont les descendants sont les arbres de e_1 et e_2, comme sur la figure 2.2a
— $t(e_1 + e_2)$ renvoie un noeud + dont les descendants sont les arbres de e_1 et e_2, comme sur la figure 2.2b

— $t(e_1^*)$ renvoie un noeud $*$ avec un seul descendant, l'arbre de e_1, comme sur la figure 2.2c



(a) Concaténation (b) Disjonction (c) Etoile

FIGURE 2.2 – Analyse syntaxique récursive de *regex*

Les règles décrites ci-avant ne disent pas comment parser une expression comme $a.b + c$. En effet, rien ne dit si elle doit être lue comme $(ab) + c$ ou $a.(b + c)$. Pour éviter d'avoir à mettre des parenthèses absolument partout, on va devoir définir les priorités entre les différentes opérations :

$$+ < . < *$$

FIGURE 2.3 – Priorités pour les opérateurs d'expressions rationnelles

Concrètement, $+ < .$ veut dire qu'une expression comme $a.b + c$ doit être interprétée comme $(a.b) + c$ [2]. De même, $a.b^*$ se lit $a.(b^*)$, et $a + b^*$ comme $a + (b^*)$.

Maintenant qu'on a les règles de dérivation et les priorités associées, on peut commencer à jouer avec quelques exemples.

Exemple 2.2.1. *L'expression rationnelle $(aa)^* + (bb)^*$ peut être parsée comme*

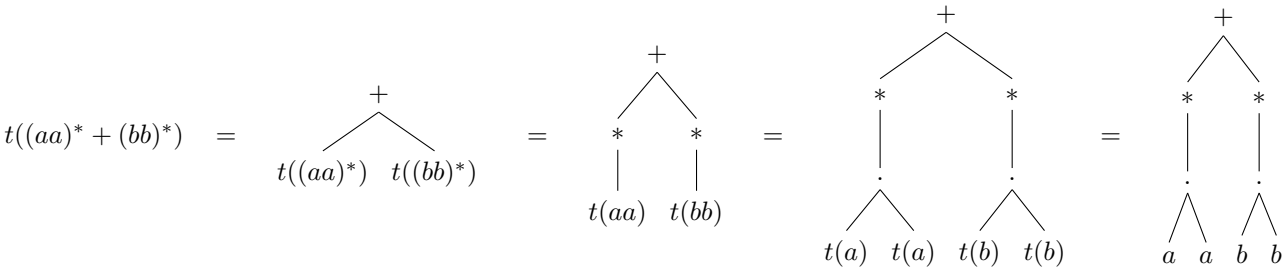
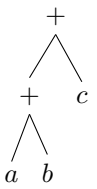


FIGURE 2.4 – Analyse syntaxique de $(aa)^* + (bb)^*$

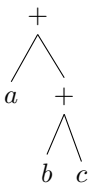
On remarquera bien sûr l'absence habile dans l'exemple 2.2.1 de formes encore problématiques : $a + b + c$ et $a.b.c$. En effet, rien ne nous dit pour l'instant auquel des arbres de la figure 2.5 la première regex correspond.

Comme on le verra dans la partie sémantique, les symboles $+$ et $.$ sont **associatifs**, ce qui veut dire que, pour toutes expressions e_1, e_2 et e_3, $(e_1 + e_2) + e_3$ et $e_1 + (e_2 + e_3)$ ont le même sens [3], et pareil avec la concaténation. Malgré une méfiance justifiée des arbres ternaires et plus, on se permettra donc d'écrire des expressions ambiguës comme $e_1 + e_2 + e_3$ ou $e_1e_2e_3$, et de les parser comme dans la figure 2.6

2. De la même façon que $a \times b + c$ se comprend comme $(a \times b) + c$
3. Notez qu'en arithmétique, $+$ et $\times$ sont également associatives



(a) version $(a + b) + c$



(b) version $a + (b + c)$

FIGURE 2.5 – Ambiguïté syntaxique de $a + b + c$

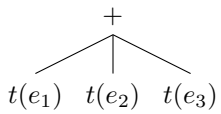
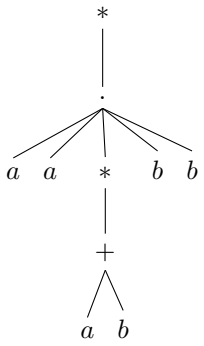


FIGURE 2.6 – Ambiguïté syntaxique assumée de $a + b + c$

Exemple 2.2.2. *L'expression rationnelle* $(aa(a + b)^*bb)^*$ *s'analyse comme*



On a pour l'instant uniquement défini le lexique et la syntaxe des expressions régulières, mais les beaux arbres qu'on est désormais en mesure de constuire n'ont en soi aucun sens, et donc aucun intérêt. Il s'agit donc désormais d'en définir une sémantique.

2.3 Sémantique

Avant de regarder la tuyauterie d'une fonction, il s'agit d'en définir le type. La sémantique des expressions rationnelles, notée $[\![e]\!]$, prend en argument une expression et renvoie un langage, donc un ensemble de mots. Comme pour le parsing, il suffit de définir la sémantique sur les 5 constructeurs des expressions rationnelles pour pouvoir toutes les traiter :

2.3.1 Les cas de base

Ici, pas de surprise, $[\![\epsilon]\!] = \{\epsilon\}$ et $[\![a]\!] = \{a\}$.

2.3.2 Sémantique de la concaténation

La sémantique de la concaténation repose sur un produit d'ensembles avec la concaténation (cf. définition A.2.1). Formellement, on a

$$\left[\!\!\left[\begin{array}{c} \cdot \\ \diagup\ \diagdown \\ e_1 \quad e_2 \end{array} \right]\!\!\right] = [\![e_1]\!].[\![e_2]\!] = \{u.v \mid u \in [\![e_1]\!] \ \wedge \ v \in [\![e_2]\!]\} = \bigcup_{\substack{u \in [\![e_1]\!] \\ v \in [\![e_2]\!]}} u.v$$

Concrètement, ça veut dire que la sémantique de la concaténation de deux regex est la concaténation des sémantiques de e_1 et e_2, c'est-à-dire l'ensemble des combinaisons d'un mot de $[\![e_1]\!]$ concaténé à un mot de $[\![e_2]\!]$. La notation $\bigcup_{\substack{u \in [\![e_1]\!] \\ v \in [\![e_2]\!]}} u.v$ est analogue une double boucle sur les éléments de $[\![e_1]\!]$ et $[\![e_2]\!]$, comme dans le pseudocode python suivant :

```
s = set()
for u in e1:
    for v in e2:
        s.add(u.v)
return s
```

Exemple 2.3.1. *En appliquant les règles de la concaténation et des lettres vues précédemment, la sémantique de l'expression ab est*

$$\left[\!\!\left[\begin{array}{c} \cdot \\ \diagup\ \diagdown \\ \boldsymbol{a} \quad \boldsymbol{b} \end{array} \right]\!\!\right] = \bigcup_{\substack{u \in [\![a]\!] \\ v \in [\![b]\!]}} u.v = \bigcup_{\substack{u \in \{a\} \\ v \in \{b\}}} u.v = \{a.b\} = \{ab\}$$

L'exemple n'est pas renversant, mais permet d'illustrer l'aspect purement systémique et récursif de la sémantique. Pour des exemples plus intéressants, on va avoir besoin d'ajouter des constructeurs à la sémantique.

2.3.3 Sémantique de la disjonction

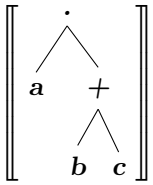
Formellement, on a

$$\left[\!\!\left[\begin{array}{c} + \\ \diagup\ \diagdown \\ e_1 \quad e_2 \end{array} \right]\!\!\right] = [\![e_1]\!] \cup [\![e_2]\!]$$

Concrètement, ça veut dire que la sémantique de la disjonction de deux regex est l'union des sémantiques de e_1 et e_2, c'est-à-dire l'ensemble des mots qui apparaissent dans $[\![e_1]\!]$ ou (inclusif) $[\![e_2]\!]$.

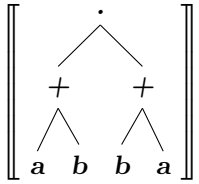
Remarque A partir d'ici et pour des raisons de mise en page, on ne mettra pas forcément tout sous forme d'arbres dans les exemples, et on comptera sur la capacité du lecteur ou de la lectrice à *parser* automatiquement toute expression rationnelle qu'il ou elle lit. Ne vous y trompez pas cependant : l'analyse sémantique s'opère bien sur un arbre plutôt que sur une expression "plate".

Exemple 2.3.2. *En appliquant les règles de la concaténation et des lettres vues précédemment, la sémantique de l'expression $a(b+c)$ est*



$$= \bigcup_{\substack{u\in[\![a]\!] \\ v\in[\![b+c]\!]}} u.v = \bigcup_{\substack{u\in\{a\} \\ v\in[\![b]\!]\cup[\![c]\!]}} = \bigcup_{\substack{u\in\{a\} \\ v\in\{b,c\}}} u.v = \{a.b, a.c\} = \{ab, ac\}$$

Exemple 2.3.3. *En appliquant les règles de la concaténation et des lettres vues précédemment, la sémantique de l'expression $(a+b)(b+a)$ est*



$$= \bigcup_{\substack{u\in[\![a+b]\!] \\ v\in[\![b+a]\!]}} u.v = \bigcup_{\substack{u\in[\![a]\!]\cup[\![b]\!] \\ v\in[\![b]\!]\cup[\![a]\!]}} = \bigcup_{\substack{u\in\{a,b\} \\ v\in\{b,a\}}} u.v = \{a.b, a.a, b.b, b.a\} = \{ab, aa, bb, ba\}$$

Il ne nous manque maintenant que le plus ésotérique des constructeurs.

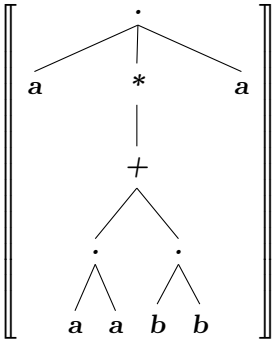
2.3.4 Sémantique de l'itération

Formellement, on a

$$[\![e^*]\!] = [\![e]\!]^* = \bigcup_{n\in\mathbf{N}} [\![e]\!]^n = \{[\![e]\!]^0, [\![e]\!]^1, [\![e]\!]^2, [\![e]\!]^3 ...\}$$

Concrètement, on fait l'union de $[\![e]\!]^n$ pour tous les entiers n, $[\![e]\!]^n$ étant n mots de $[\![e]\!]$ concaténés.

Exemple 2.3.4. *La sémantique de l'expression $a(aa+bb)^*a$ est*



$$= [\![a]\!].[\![(aa+bb)^*]\!].[\![a]\!]$$

$$= \{a\}.\bigcup_{n\in\mathbf{N}}[\![aa+bb]\!]^n.\{a\}$$

$$= \{a\}.\bigcup_{n\in\mathbf{N}}\{aa, bb\}^n.\{a\}$$

$$= \{a.\epsilon.a, a.aa.a, a.bb.a, a.((aa).(aa)).a, a.((aa).(bb)).a, a.((bb).(aa)).a, a.((bb).(bb)).a, ...$$

$$= \{aa, aaaa, abba, aaaaaa, aaabba, abbaaa, abbbba, ...\}$$

2.4 Mise en application

On a abordé les expressions régulières sous un angle très théorique, mais on leur trouve bien sûr des applications concrètes.

2.4.1 Quelques astuces

On présente d'abord, sous forme d'exercices (corrigés dans un autre document), quelques astuces classiques, susceptibles d'aider les TAListes dans leurs futures oeuvres.

Exercice 2.4.1. *Donner une regex pour les mots qui commencent par a.*

Exercice 2.4.2. *Donner une regex pour les mots qui finissent par b.*

Exercice 2.4.3. *Donner une regex pour les mots qui commencent par a finissent par b.*

Exercice 2.4.4. *Donner une regex pour les mots de longueur paire.*

Exercice 2.4.5. *Donner une regex pour les mots de longueur impaire qui contiennent au moins 4 lettres.*

Exercice 2.4.6. *Donner une regex pour les mots de longueur impaire, qui contiennent au moins 4 lettres, commencent par a et finissent par b.*

2.4.2 Syntaxe en pratique

Les expressions régulières dans Unix, Python & cie utilisent une syntaxe différente, et surtout plus étendue que celle que l'on vient d'étudier. Celà est dû aux besoins différents que l'on a entre la théorie et la pratique.

Dans la théorie, on veut définir nos objets de façon minimale, c'est-à-dire avec le moins de symboles et de règles possible, afin d'en simplifier l'étude. Par exemple, le peu de règles permet une définition légère de la sémantique formelle des *regex*. De la même façon, toute preuve à leur propos en sera tout autant simplifiée :

Théorème 1. $\forall e, \exists w \in [\![e]\!]$, *cad. que toute expression rationnelle dénote au moins un mot.*

Démonstration. On procède par induction structurelle sur l'expression rationnelle e :

- Si $e = \epsilon$, alors $[\![e]\!] = \{\epsilon\}$, qui contient bien un mot (ϵ donc)
- Si $e = a$, alors $[\![e]\!] = \{a\}$, qui contient bien un mot (a)
- Si $e = e_1 + e_2$, alors $[\![e]\!] = [\![e_1]\!] \cup [\![e_2]\!]$. Par hypothèse d'induction, $[\![e_1]\!]$ contient un mot w_1 et $[\![e_2]\!]$ contient w_2. $[\![e]\!]$ contient donc non pas un, mais au moins deux mots.
- Si $e = e_1.e_2$, alors $[\![e]\!] = [\![e_1]\!].[\![e_2]\!]$. Par hypothèse d'induction, $[\![e_1]\!]$ contient un mot w_1 et $[\![e_2]\!]$ contient w_2. $[\![e]\!]$ contient donc un mot, $w_1 w_2$.
- Si $e = e_1^*$, alors $[\![e]\!]$ contient ϵ.

□

Dans la pratique, on préfère ne pas avoir à réinventer la roue chaque matin, la syntaxe des *regex* y est donc étendue. Ces extensions ne changent rien au fond, dans le sens où elles n'ajoutent pas en expressivité. En effet, les nouveaux symboles peuvent tous être codés avec ceux de la syntaxe minimale :

- $e?$, ou "e une ou zéro fois", peut être codée comme $(e + \epsilon)$

— $e+$, ou "e au moins une fois", peut être codée comme ee^*
remarque On trouve régulièrement cette notation dans la littérature académique sous la forme e^+. Le $e_1 + e_2$ qu'on a vu est lui, pour le coup, écrit $e_1|e_2$ en pratique.
— $e\{n\}$, ou "e exactement n fois" peut être simplement traduite en $\underbrace{eee...eee}_{n \text{ fois}}$
— $e\{n,\}$, ou "e au moins n fois" peut être simplement traduite en $\underbrace{eee...eee}_{n \text{ fois}}e^*$
— $e\{n,m\}$, ou "e entre n et m fois" peut se traduire $\underbrace{eee...eee}_{n \text{ fois}}\underbrace{(e+\epsilon)...(e+\epsilon)}_{m-n \text{ fois}}$

Les traductions proposée ici ne correspondent pas forcément à ce qui se passe concrètement dans les bibliothèques de *regex* des différents langages de programmation (la dernière en particulier semble très ambigüe, et donc inefficace). L'objectif est seulement de montrer que les ajouts à la syntaxe n'ent modifient pas l'expressivité, et qu'il s'agit seulement de ce qu'on appelle du **sucre syntaxique**.

2.4.3 Calcul de l'appartenance

Soit l'expression rationnelle $e = ((\Sigma^* baaba^+)^* baa(abba)^+ba(bb)^*a)^*$, il n'est (normalement) pas totalement évident de déterminer automatiquement si un mot donné appartient à sa sémantique (on pourra se convaincre en essayant à la main avec par exemple *baabbaaabaababbaaa*). Or, pour mettre en application les expressions rationnelles [4], on va avoir besoin d'un tel algorithme. On pourrait par exemple instancier e de toutes les façons possibles en bornant les étoiles par la longueur du mot donné, mais la complexité d'une telle procédure n'est pas réaliste. Un algorithme plus raisonnable va nous être fourni via les **automates finis**.

4. Notamment pour en repérer des occurences dans du texte

Chapitre 3

Automates finis

Les automates forment un langage de programmation un peu particulier, en ce qu'il est très visuel (chaque programme est un graphe annoté, ou plus prosaïquement des ronds et des flèches) et que tout programme a le même type : un mot en entrée, un booleen en sortie. Un automate définit donc un langage en donnant un moyen automatique de déterminer si n'importe quel mot donné en fait partie ou non [1].

Les automates se divisent en de nombreuses sous-catégories, dont certaines ramifications seront explorées dans ce cours. On verra d'abord le fonctionnement général des automates finis déterministes (3.1) et non-déterministes (3.2), en allant à chaque fois du général au technique. On verra ensuite des algorithmes pour transformer (3.3) des automates. On étudiera enfin les combinaisons d'automates, et donc les propriétés de clôture des langages qu'ils définissent (3.5).

3.1 Automates finis déterministes

On introduit les Automates finis déterministes, noté AFD (ou DFA, pour *deterministic finite automaton* [2]), avant d'en étudier la formalisation.

3.1.1 Principe général

Imaginez que vous développiez un jeu d'infiltration. Dans ce jeu, le comportement des méchants gardes ressemblerait sans doute à la figure 3.1.

Ce qu'est censé traduire ce graphe, c'est qu'un garde commence (la $\rightarrow$ sur sa gauche) dans un **état** qui est le dodo, et que différents événements (un bruit, un ennemi trouvé ou non, ou encore tué) vont le faire changer d'**état**. Un AFD fonctionne sur un principe similaire [3], mais où les transitions sont déclenchées par la lecture de lettres : un AFD, en partant d'un état initial, lit le mot donné en argument lettre par lettre et, à chaque lecture, change d'état en fonction de la lettre.

1. En ce sens, les automates sont des fonctions caractéristiques, qui sont aux ensembles ce que les videurs sont aux boîtes de nuit.
2. Notez qu'on parle d'*automat**on*** au singulier et d'*automat**a*** au pluriel
3. Les AFD sont en fait des cas particuliers de Machines à états finis, qui sont effectivement employées dans la conception de jeux vidéo

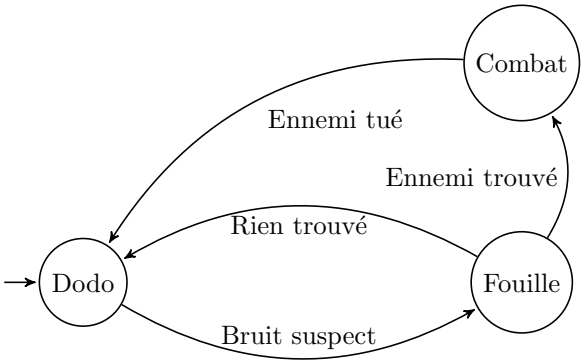


FIGURE 3.1 – Comportement des gardes d'un jeu imaginaire

Exemple 3.1.1. *L'automate de la figure 3.2 contient trois états, appelés* 0, 1 *et* 2. *La lecture de tout mot commence en* 0, *appelé* ***état initial****. Si on lui passe le mot abbaaba en argument, la première lettre (a) va nous faire passer de l'état* 0 *à* 1. *La deuxième lettre (b) nous refait passer en* 0. *La troisième (b) nous y fait rester. Les deux lettres suivantes nous font ensuite passer en* 1 *puis en* 2. *Les deux dernières lectures nous font rester en* 2 *(la virgule est à comprendre comme une disjonction, cad. comme un "ou").*

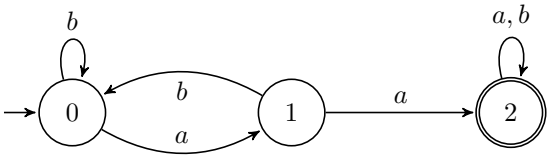


FIGURE 3.2 – Un premier automate

Comme dit en introduction, un automate accepte ou rejette tout mot donné. Certains états, notés par une douche couche, sont appelés états finaux (ou terminaux). Un mot est accepté par un automate si et seulement si le parcours de ce mot dans l'automate se termine sur un état final.

Exemple 3.1.2. *L'automate de la figure 3.2 accepte le mot abbaaba, puisqu'il nous fait passer de l'état initial* 0 *à* 2, *qui est un état final. Il n'accepte en revanche pas les mots bbaba (état* 1*), babbab ou* ϵ *(état* 0 *tous les deux).*

Exercice 3.1.1. *Les mots abbaba, ababbaab et abba sont-ils acceptés par l'automate de la figure 3.2 ?*

Exercice 3.1.2. *Quel est le* ***langage reconnu****, cad. l'ensemble des mots acceptés, par l'automate de la figure 3.2 ? Donner la réponse en français et sous forme d'expression rationnelle.*

Remarque Un automate ne contient pas toujours une transition pour chaque couple d'état / lettre, auquel cas il est dit **incomplet**. Si un automate ne contient pas de chemin correspondant à un mot, ce dernier est rejetté.

Exemple 3.1.3. *L'automate de la figure 3.3 rejette le mot aba, car il n'y a pas de transition partant de l'état* 1 *pour la lettre* b.

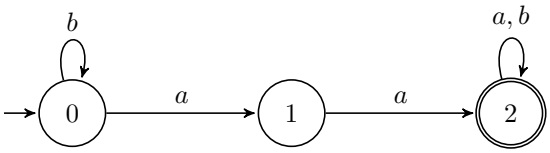


FIGURE 3.3 – Un automate incomplet

Exercice 3.1.3. *Les mots bbbaababbaaba, bbabaab et baaaaaab sont-ils acceptés par l'automate de la figure 3.3 ?*

Exercice 3.1.4. *Quel est le langage reconnu par l'automate de la figure 3.3 ? Donner la réponse en français et sous forme d'expression rationnelle.*

Il nous semble important d'insister sur le point suivant : de la même façon qu'un programme devrait la traduction d'une logique sous-jacente plutôt qu'un bidouaille fait à la va-vite, les états d'un automate ont un sens. Avant d'écrire un automate, il convient donc de réfléchir quelles sont les informations à retenir au cours de la lecture du mot. Si la bonne réponse est trouvée, le reste de l'automate devrait s'écrire seul.

Exemple 3.1.4. *On veut écrire un automate reconnaissant le langage* $L = \{w \in \Sigma^* \mid |w|_a$ *pair et* $|w|_b$ *impair*$\}$, *cad. l'ensemble des mots avec un nombre pair de* a *et impairs de* b[4].

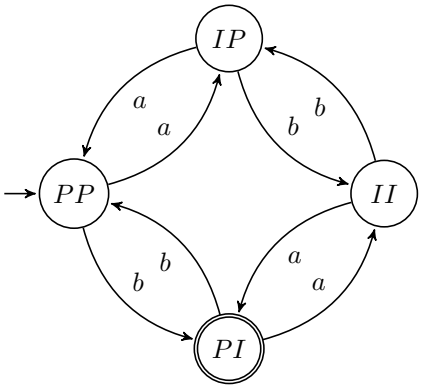
Il n'est pas question de compter les a *et les* b *comme on pourrait naïvement l'imaginer, non seulement puisqu'il faut se contenter d'un nombre fini d'états, mais aussi parce que c'est beaucoup plus d'information que nécessaire. Les seules données qui nous intéressent sont en effet la parité du nombre de* a *et du nombre de* b *du mot donné :* a^2b *comme* $a^{26}b^{131}$ *sont équivalents dans leur appartenance à* L.

Le nombre de a *et de* b *étant tous les deux pairs ou impairs, on a 4 possibilités. Nos états s'appeleront* PP *(nombre de* a *pair et nombre de* b *pair),* PI *(*a *pair et* b *impair),* IP *et* II. *La définition de* L *nous dit immédiatement que seul* PI *devrait être terminal. L'état initial devrait être celui qui correspond à* ϵ, *cad.* PP.

Les transitions s'écrivent naturellement : en partant de PP, *la lecture d'un* a *change la parité du nombre de* a *mais pas celle du nombre de* b, *et nous emmène donc vers* IP, *tandis que* b *pointe vers* PI, *et ainsi de suite. S'il y a d'autres lettres dans l'alphabet, elles devraient faire des boucles, puisqu'elles ne changent rien aux parités qui nous intéressent.*

Au final, on obtient l'automate suivant :

4. On conseillera tout d'abord au lecteur ou à la lectrice de tenter lui/elle-même l'exercice, afin de mesurer la pertinence de l'approche ici présentée



Exercice 3.1.5. *(**) En reprenant l'exemple 3.1.4, montrer que* $\forall w, w \in L \leftrightarrow$ *l'automate accepte* w*. Vous pouvez procéder par induction sur* w*, en utilisant un objectif un peu plus précis que celui fourni.*

Dans la série d'exercices qui suit, on utilisera comme alphabet $\Sigma = \{a, b\}$.

Exercice 3.1.6. *Donner un automate qui reconnaît le langage* $\{w \in \Sigma^* \mid |w| \geq 3\}$.

Exercice 3.1.7. *Donner un automate pour les mots qui commencent par* a.

Exercice 3.1.8. *Donner un automate pour les mots qui finissent par* b.

Exercice 3.1.9. *Donner un automate pour les mots qui commencent par* a *finissent par* b.

Exercice 3.1.10. *Donner un automate pour les mots de longueur paire.*

Exercice 3.1.11. *Donner un automate pour les mots de longueur impaire qui contiennent au moins 4 lettres.*

Exercice 3.1.12. *Donner un automate pour les mots de longueur impaire, qui contiennent au moins 4 lettres, commencent par* a *et finissent par* b.

3.1.2 Formalisation et implémentation

Un automate fini déterministe est un $5-$*uplet* (cad. un "paquet" qui contient 5 éléments ordonnés)

$$\langle Q, \Sigma, q_0, F, \delta \rangle$$

avec

Q ensemble fini d'états
Σ l'alphabet
q_0 l'état initial
$F \subseteq Q$, les états terminaux

δ fonction partielle [5] de $Q \times \Sigma$ dans Q

La fonction δ est alors *liftée* aux mots [6] pour obtenir la fonction $\delta^* : (Q \times \Sigma^*) \to Q$ définie de la façon suivante :

— $\delta^*(q, \epsilon) = q$
— $\delta^*(q, a.w) = \delta^*(\delta(q, a), w)$ (a est une lettre et w un mot, éventuellement vide)

Plus prosaïquement, la fonction δ^* applique δ sur w, lettre par lettre, de la gauche vers la droite. Dans ce cadre, on dit qu'un mot w est **reconnu** (ou accepté) par l'automate ssi. $\delta^*(q_0, w) \in F$. Dit encore autrement, un mot est accepté ssi en partant de l'état initial de l'automate et en suivant les transitions correspondant aux lettres successives du mot on finit dans un état terminal.

La définition de δ^* donne quasiment directement une implémentation de la reconnaissance via automate fini déterministe :

```
state = q_0
for c in w:
    state = delta(state,c)
return (state in F)
```

FIGURE 3.4 – Reconnaissance par un AFD

Exemple 3.1.5. *L'automate de la figure 3.2 se formalise de la façon suivante :*

— $Q = \{0, 1, 2\}$
— $\Sigma = \{a, b\}$
— $q_0 = 0$
— $F = \{2\}$
— $\delta(0, a) = 1; \delta(0, b) = 0; \delta(1, a) = 2; \delta(1, b) = 0; \delta(2, a) = 2; \delta(2, b) = 2$

On peut vérifier qu'il accepte bien le mot abbaaba (lecture comme une BD) :

$$\begin{array}{rcll} \delta^*(0, abbaaba) & = & \delta^*(\delta(0, a), bbaaba) & = & \delta^*(1, bbaaba) \\ & = & \delta^*(\delta(1, b), baaba) & = & \delta^*(0, baaba) \\ & = & \delta^*(\delta(0, b), aaba) & = & \delta^*(0, aaba) \\ & = & \delta^*(\delta(0, a), aba) & = & \delta^*(1, aba) \\ & = & \delta^*(\delta(1, a), ba) & = & \delta^*(2, ba) \\ & = & \delta^*(\delta(2, b), a) & = & \delta^*(2, a) \\ & = & \delta^*(\delta(2, a), \epsilon) & = & \delta^*(2, \epsilon) \\ & & & = & 2 \in F \end{array}$$

Exercice 3.1.13. *Donner la formalisation de l'automate de l'exemple 3.1.4 et vérifier qu'il accepte le mot aababba.*

On a introduit en 2.4.3 l'expression rationnelle $e = ((\Sigma^* baaba^+)^* baa(abba)^+ ba(bb)^* a)^*$, qu'on était censé traduire en programme via les automates finis. Or, la traduction en AFD n'est à

5. Une fonction partielle est une fonction qui n'attribue pas forcément une image à tout argument. Dans le cas d'un automate complet, la fonction de transition est donc une fonction totale.

6. On dit qu'une fonction est *liftée* lorsqu'on "soulève" le type d'un ou plusieurs de ses arguments, pour qu'elle s'applique par exemple à des listes ou des ensembles. Typiquement, si on a une fonction f de type $\mathbb{N} \to \mathbb{N}$, alors sa version *liftée* aux listes, étant donnée une liste $[a_1, ..., a_n]$, renverra $[f(a_1), ..., f(a_n)]$.

priori pas si évidente que ça (on conseille à nouveau d'essayer pour se rendre compte de la difficulté), du fait du haut niveau de non-déterminisme dans la *regex*. On va donc avoir besoin d'un modèle un peu plus permissif.

3.2 Automates finis non-déterministes

Les automates finis qu'on a vus jusqu'ici sont déterministes, en ce qu'un automate admet au maximum une seule transition par lettre, et donc que tout mot n'a lui-même pas plus d'un chemin. On va ici voir une nouvelle classe d'automates, les automates finis non-déterministes (AFND, ou *NDFA* pour *non-deterministic finite automaton/a*), qui vont justement nous libérer de cette contrainte.

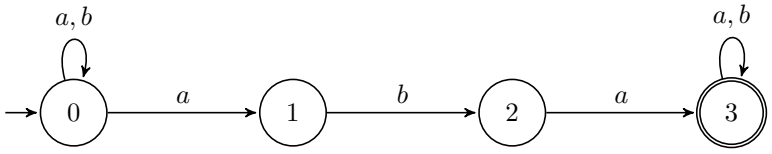
3.2.1 Principe général

Le non-déterminisme se manifeste de trois façons : les ϵ-transitions, qui sont des transitions "gratuites", la possibilité d'avoir plusieurs transitions pour le même couple état $\times$ lettre, et celle d'avoir plusieurs états initiaux.

Les ϵ-transitions sont présentées et illustrées dans le premier DM. Ce dernier présente notamment un algorithme permettant de les éliminer, cad. qui prend en entrée un automate et en génère un acceptant le même langage sans ϵ-transitions. Les ϵ-transitions sont donc une commodité qu'on peut au choix ignorer ou utiliser. Pour ne pas alourdir le présent document, on l'y ignorera, à l'exception des sous-sections 3.5.2 et 3.5.3, significativement simplifiées par l'utilisation d'ϵ-transitions.

Concentrons nous sur les choix de transitions et d'états initiaux. On peut maintenant avoir dans un automate plusieurs chemins pour un même mot, chemins qui vont chacun mener ou non à un état terminal. On accepte tout mot qui mène à un état terminal via au moins un chemin.

Exemple 3.2.1. *Le langage des mots qui contiennent le facteur aba, dénoté par la regex* $\Sigma^* aba \Sigma^*$*, est reconnu par l'automate suivant :*



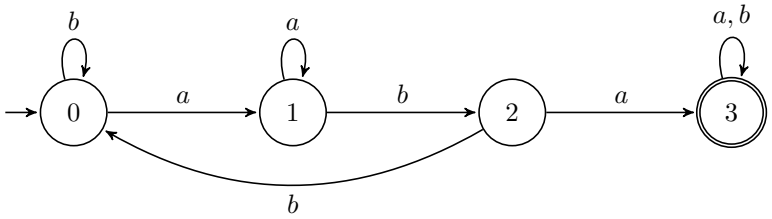
Si on regarde par exemple le mot aabab, il dispose de 3 parcours dans l'automate :

- $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{b} 0$
- $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 1 \xrightarrow{b} 2$
- $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 3$

On peut aussi passer dans l'état 1 *avec le premier a, mais il sera alors impossible de lire l'entiéreté du mot (le deuxième a n'aura pas de transition).*

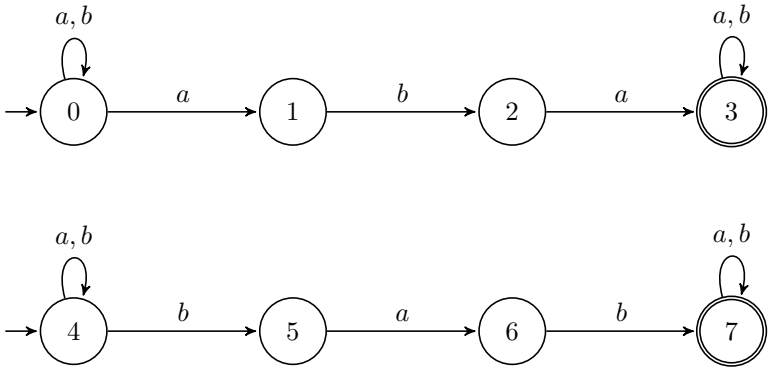
Des trois chemins possibles, un seul mène à un état terminal. C'est néanmoins suffisant pour que le mot soit accepté. A l'inverse, le mot abbaabbaaabba, bien que pouvant faire de très nombreux chemins dans l'automate, n'est pas accepté car aucun ne finit en 3

Un tel automate a l'avantage de reconnaître très clairement le langage $\Sigma^* aba \Sigma^*$. On peut par exemple le comparer à sa version déterministe qui, malgré la simplicité du langage, perd déjà pas mal en lisibilité :



La grande différence entre les deux automates est que le premier utilise le non-déterminisme pour être une transcription directe de la *regex*, là où le deuxième se bat avec le déterminisme pour reconnaître le langage de façon presque "accidentelle". En un sens, on est passé du domaine de la **spécification** à l'**implémentation**. La mauvaise nouvelle, c'est que les automates non-déterministes sont plus compliqués à mettre en pratique, du fait - ô surprise - du non-déterminisme, qui impose de tester énormément de chemins. La bonne nouvelle, c'est que, comme on va le voir en 3.3.2, on peut traduire automatiquement les automates non-déterministes en automates déterministes équivalents [7], avec bien sûr un certain coût. Mais avant de plonger dans ces histoires, on va voir encore quelques exemples d'AFND, et en étudier la formalisation.

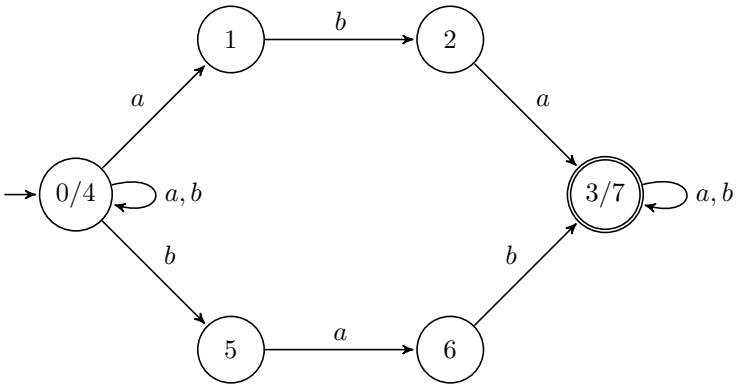
Exemple 3.2.2. *On veut reconnaître le langage des mots qui contiennent le facteur aba ou (inclusif) bab. On peut exploiter la possibilité d'avoir plusieurs états initiaux, et tout simplement produire l'automate suivant :*



On notera bien sûr la symétrie entre l'automate ci-dessus, et la regex $\Sigma^* aba \Sigma^* + \Sigma^* bab \Sigma^*$.

On peut aussi tenter d'être un poil plus malin, se rendre compte que les états 0 *et* 4 *ont le même* ***rôle****, que les états* 3 *et* 7 *aussi, et donc qu'on peut réduire le nombre d'états en les fusionnant :*

7. Pour se rendre compte du miracle que c'est, on peut comparer aux langages de programmation classiques. Ce n'est pas parce qu'on a spécifié formellement, par exemple, le tri d'une liste (ce qui est déjà surprenamment non-trivial) qu'on peut automatiquement obtenir un programme réalisant cette tâche. Même une fois qu'on a codé un tel programme, selon l'algorithme chopisi, il n'est pas forcément aisé de se convaincre qu'il est correct, et encore moins de le prouver formellement. Il est donc assez *cool* que tout ce passage de la spécification (la *regex*) à une représentation intermédiaire (l'automate non-déterministe) à l'implémentation (le déterministe) soit automatique et sûre.



Cette notion d'états équivalents et fusionnables sera formalisée en 3.3.3. On remarquera (à nouveau) la symétrie entre cet automate et la regex $\Sigma^*(aba + bab)\Sigma^*$.

Exercice 3.2.1. *Donner une regex et un automate fini pour le langage $L = \{w \mid aba$ est un sous-mot de $w\}$.*

Exercice 3.2.2. *Donner une regex et un automate fini pour le langage des mots qui commencent par ab et finissent par ba.*

3.2.2 Formalisation et implémentation

Les automates non-déterministes sont une généralisation [8] des automates déterministes, où

- L'état initial q_0 est remplacé par un ensemble d'états initiaux $I \subseteq Q$
- La fonction de transition δ change de type, en passant de $(Q \times \Sigma) \to Q$ à $(Q \times \Sigma) \to P(Q)$ [9]. Dit autrement, étant donnés un état $q \in Q$ et une lettre $c \in \Sigma$, on a (potentiellement) *accès* à un ensemble d'états plutôt qu'à un seul.

Puisqu'on a changé le type δ, on doit également changer son *lift* δ^*, qui renvoie maintenant un ensemble d'états et est donc de type $(Q \times \Sigma^*) \to P(Q)$:

- $\delta^*(q, \epsilon) = \{q\}$
- $\delta^*(q, a.w) = \bigcup_{q' \in \delta(q,a)} \delta^*(q', w)$

Il nous reste à vérifier que l'ensemble d'états atteignales contient un état terminal. Un AFND accepte donc un mot w ssi. $\exists q \in ((\bigcup_{q_i \in I} \delta^*(q_i, w)) \cap F)$.

Pour ce qui est de la mise en application de δ^*, on est cette fois face à un parcours d'arbres (l'ensemble des chemins) plutôt que d'une liste, comme dans la version déterministe. On adapte donc l'annexe A.3.1 :

8. En effet, un AFD peut-être vu comme un AFND qui se trouve être ... déterministe. Rien dans la définition d'un AFND n'impose que le non-déterminisme permis par le changement de type de δ soit exploité, et on peut donc très bien avoir $|\delta(q, c)| \leq 1\ \forall q \in Q$ et $\forall c \in \Sigma$.

9. Pour rappel, $P(Q)$ est l'ensemble des sous-parties de Q, cad. l'ensemble des ensembles d'états, cf A.2.2.

```
# fonction ndfa_acc(auto,w)
# renvoie true ssi. l automate auto accepte le mot w
todo = stack()
// On va empiler des couples etat x mot qu il faut tester
// On commence par se noter tous les etats initiaux
for i in initial(auto):
    todo.add(i,w)
while (todo pas vide):
    (q,mot) = todo.pop()
    // Si on a fini de lire le mot et qu on est
    // arrive sur un etat final, on accepte
    // Si l etat n est pas final, on ne renvoie
    // pas false, car il peut rester des chemins
    // qu il faut tester
    if mot is empty et q in F:
        return true
    elif mot is a.reste:
        // On continue la lecture du mot avec
        // chaque etat qu on peut atteindre
        // avec la premiere lettre
        next_states = delta(q,a)
        for suivant in next_states:
            todo.add(suivant,reste)
// Si on n a au final rien trouve, on dit non
return false
```

FIGURE 3.5 – Reconnaissance par un AFND - version iterative

```
# fonction ndfa_acc_bis(auto,w,q)
# renvoie true ssi. l automate auto accepte le mot w en partant de l etat q
if w is empty:
    return q in F
elif w is a.reste:
    for suivant in delta(q,a):
        if ndfa_acc_bis(auto,reste,suivant):
            return true
    return false


# fonction ndfa_acc(auto,w)
# renvoie true ssi. l automate auto accepte le mot w
for i in initial(auto):
    if ndfa_acc_bis(auto,w,i):
        return true
return false
```

FIGURE 3.6 – Reconnaissance par un AFND - version récursive

3.3 Transformation d'automates

La formalisation très limitée et simple des automates en fait des programmes particulièrement simple à manipuler, comme l'illustrent les algorithmes de transformation d'automates présentés ici.

3.3.1 Complétion

On peut facilement compléter un automate en rajoutant un état "poubelle", non terminal et bouclant, qui absorbera tous les transitions ajoutées.

Exemple 3.3.1. *Si on complète l'automate de la figure 3.3, on rajoute un état P qui boucle sur lui-même. Il manquait dans l'automate uniquement une b-transition partant de l'état 1. On rajoute donc $1 \xrightarrow{b} P$, et on obtient*

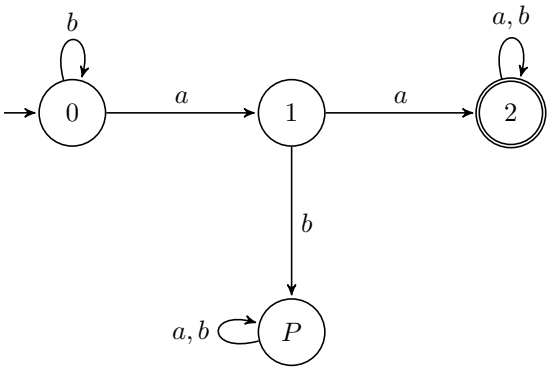


FIGURE 3.7 – Un automate complété

Exercice 3.3.1. *Compléter le premier automate de l'exemple 3.2.2.*

On peut se convaincre assez facilement que la complétion ne change pas le langage reconnu par un automate : un parcours prend de nouvelles transitions ssi. il ne pouvait aller jusqu'au bout dans l'automate initial. Dans les deux cas, le mot associé n'est pas accepté, puisqu'on boucle sur un état non-terminal dans le premier cas, et on "plante" dans le second.

Cette transformation se formalise également assez facilement. Soit un automate déterministe $\langle Q, \Sigma, q_0, F, \delta \rangle$, alors on renvoie l'AFD $\langle Q \cup \{P\}, \Sigma, q_0, F, \delta' \rangle$, avec

$$\begin{cases} \delta'(q,a) = \delta(q,a) & \text{si } \delta(q,a) \text{ défini,} \\ \delta'(q,a) = P & \text{si } \delta(q,a) \text{ non défini,} \\ \delta'(P,a) = P & \text{pour tout } a \in \Sigma \end{cases}$$

Notez qu'on met les cas $\delta'(P,a)$ à part, car (on suppose) $P \notin Q$, ce qui implique que $\delta(P,a)$ n'a pas de sens.

Exercice 3.3.2. *Donner la formalisation de la complétion d'un automate non-déterministe.*

3.3.2 Déterminisation

L'algorithme présenté ici prend en entrée un AFND et renvoie un AFD reconnaissant le même langage. L'idée est que l'AFD va simuler le comportement de l'AFND, et donc, étant donné un mot, que le (seul) parcours dans l'AFD va représenter tous ceux dans l'AFND.

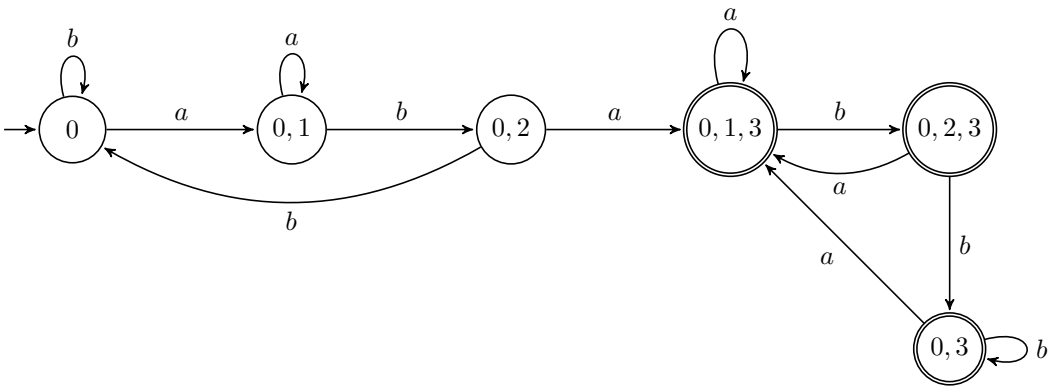
Pour ça, l'ensemble d'états de l'AFD va être l'ensemble des combinaisons d'états de l'AFND. Par exemple, si on a dans ce derniers deux états initiaux i_1 et i_2, l'automate initial de l'AFD sera l'état $\{i_1, i_2\}$. Supposons de plus que les transitions partant de i_1 et i_2 sont les suivantes :

— $i_1 \xrightarrow{a} q_1$
— $i_1 \xrightarrow{a} q_2$
— $i_2 \xrightarrow{a} q_1$
— $i_2 \xrightarrow{a} q_3$

Tout mot commençant par a lu par l'AFND peut donc faire son premier saut en q_1, q_2 ou q_3. On ajoute donc à l'AFD la transition $\{i_1, i_2\} \xrightarrow{a} \{q_1, q_2, q_3\}$, et ainsi de suite.

Arriver, après lecture d'un mot, dans un état X de l'AFD signifie donc que, dans l'AFND, on peut atterir dans chacun des états contenus dans X après la lecture du même mot. Puisqu'un AFND accepte quand au moins un des parcours termine sur un état terminal, tout état de l'AFD contenant un état terminal est lui-même terminal.

Exemple 3.3.2. *Si on essaye de déterminiser l'automate de l'exemple 3.2.1, on obtient*



On notera une grande proximité avec la déterminisation "à la main" qu'on avait initialement faite dans l'exemple. En effet, on a les correspondances suivantes entre les états :

— $0 \approx 0$
— $1 \approx 0, 1$
— $2 \approx 0, 2$
— $3 \approx 0, 1, 3 + 0, 2, 3 + 0, 3$

Exercice 3.3.3. *Déterminiser le premier automate de l'exemple 3.2.2. (la correction de cet exercice contient, contrairement à l'exemple ci-dessus, quelques étapes intermédiaires).*

La déterminisation se formalise aussi assez bien en termes de définitions. Soit un automate $\langle Q, \Sigma, I, F, \delta \rangle$, alors on renvoie l'automate $\langle P(Q), \Sigma, \bigcup_{i \in I} i, F', \delta' \rangle$, avec

$$\delta'(X, a) = \bigcup_{q \in X} \delta(q, a)$$

$$F' = \{X \in P(Q) \mid X \cap F \neq \emptyset\}$$

Notez que suivre cette définition génère un automate qui contient tout un tas d'états qui vont être inatteignables, et donc ne servir à rien. En l'appliquant sur l'exemple 3.3.2, on devrait notamment produire les états $\{1, 2\}$ ou $\{2, 3\}$. En pratique, on préfèrera donc suivre l'algorithme présenté avec les mains, qui consiste à partir de l'état initial et d'ajouter les états / transitions au fur à mesure, jusqu'à atteindre un point fixe.

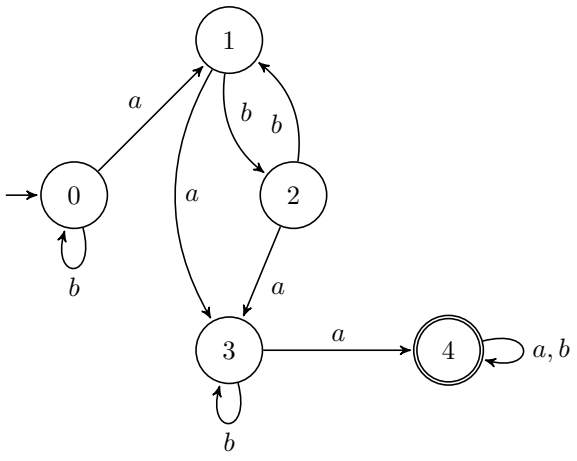
Il s'agit d'un résultat extrêmement important de la théorie des automates, puisqu'il établit l'équivalence entre automates finis déterministes et non-déterministes. Pour montrer qu'un langage est reconnaissable par un AFD, on pourra se contenter de fournir un AFND. D'un point de vue plus pratique, on pourra abondamment utiliser le non-déterminisme dans nos conceptions d'automates, ce dernier pouvant être éliminé par cet algorithme. Cette transformation a cependant un coût, qui peut être très élevé. En effet, le lemme 12 implique que la déterminisation d'un automate à n états peut en avoir jusqu'à 2^n.

On peut commencer à voir pourquoi en re-regardant l'exemple 3.3.2, où on a 3 états terminaux alors qu'un seul serait suffisant : une fois qu'on a atteint l'état $\{0, 1, 3\}$, on ne fait que tourner entre trois états terminaux, ce qui veut dire que n'importe quel (reste de) mot sera accepté. L'algorithme de déterminisation ne s'intéresse pas au rôle des états (il est **syntaxique**, et non **sémantique**), et ne voit donc pas qu'une fusion est ici possible. De telles optimisations vont cependant nous être fournies par le prochain algorithme.

3.3.3 Minimisation

On a dit dans l'exemple 3.3.2 que plusieurs états avaient le même rôle et pouvaient être fusionnés. On va ici présenter un algorithme qui prend en entrée un automate déterministe complet [10] et en classe les états selon leur rôle, ce qui permet ensuite de fusionner à l'intérieur de chaque classe. On obtient alors un automate reconnaissant le même langage que celui en entrée, mais de façon minimale (par rapport au nombre d'états). On va présenter l'algorithme *via* l'exemple suivant :

Exemple 3.3.3. *Soit l'automate suivant :*



On disait plus haut qu'on veut regrouper les états selon leur rôle. Le rôle d'un état q est l'ensemble des mots qu'il accepte, cad. l'ensemble des mots qui vont de q à un état terminal, ou $\{w \mid \delta^*(q, w) \in F\}$. On va partir d'un regroupement très approximatif, et affiner petit à petit.

10. Ce qui, grâce aux deux algorithmes précédents, sont des hypothèses gratuites.

Pour commencer, on peut séparer les états terminaux des non-terminaux en deux sous-ensembles, ou classes. En effet, les premiers acceptent le mot ϵ alors que les seconds non. On divise donc notre ensemble d'états en $T = \{4\}$ et $N = \{0, 1, 2, 3\}$. On ne va manifestement pas pouvoir affiner T, contrairement à N. Pour ça, on va regarder toutes les paires d'états de N et, à chaque fois, vérifier si les deux états ont des désaccords en lisant a ou b :

- *0 vs. 1*
  - $\delta(0, a) = 1$ *et* $\delta(1, a) = 3$. *Pour l'instant,* 1 *et* 3 *appartiennent tous les deux à* N*, ce qui veut dire qu'on suppose que* 1 *accepte un mot* w *ssi.* 3 *accepte* w*. On peut donc conclure que, de ce qu'on sait,* 0 *accepte un mot* $a.w$ *ssi.* 1 *accepte* $a.w$*.*
  - $\delta(0, b) = 0$ *et* $\delta(1, b) = 2$. 0 *et* 2 *appartiennent à la même classe, on suppose donc pour l'instant que* 0 *accepte un mot* $b.w$ *ssi.* 1 *accepte un mot* $b.w$*.*
  - ⇒ *Pour l'instant, on conserve notre hypothèse selon laquelle* 0 *et* 1 *acceptent le même langage.*
- *0 vs. 2*
  - $\delta(0, a) = 1$ *et* $\delta(2, a) = 3$, 1 *et* 3 *sont dans la même classe.*
  - $\delta(0, b) = 0$ *et* $\delta(2, b) = 2$. 0 *et* 2 *appartiennent à la même classe.*
  - ⇒ *Pour l'instant, on conserve notre hypothèse selon laquelle* 0 *et* 2 *acceptent le même langage.*
- *0 vs. 3*
  - $\delta(0, a) = 1$ *et* $\delta(3, a) = 4$, 1 *et* 4 *ne sont pas dans la même classe, ce qui veut dire qu'il existe un mot* w [11] *sur lequel* 1 *et* 4 *ne sont pas d'accord, dans le sens où* 1 *accepte* w *alors que* 4 *non, ou l'inverse. On en déduit donc que* 0 *et* 3 *sont en désaccord sur* $a.w$*. Puisqu'ils sont en désaccord, ils ne devraient pas être dans la même classe.*
  - ⇒ *Pas besoin de tester avec* b*, on sait déjà qu'on va séparer* 0 *et* 3*.*
- *1 vs. 2*
  - $\delta(1, a) = 3$ *et* $\delta(2, a) = 3$*, pas de souci.*
  - $\delta(1, b) = 2$ *et* $\delta(2, b) = 1$*, idem.*
  - ⇒ *On garde* 1 *et* 2 *dans la même classe.*
- *1 vs. 3*
  - $\delta(1, a) = 3$ *et* $\delta(3, a) = 4$*.*
  - ⇒ 3 *et* 4 *ne sont pas dans la même classe, on doit donc séparer* 1 *et* 3*.*
- *2 vs. 3*
  - $\delta(2, a) = 3$ *et* $\delta(3, a) = 4$*.*
  - ⇒ 3 *et* 4 *ne sont pas dans la même classe, on doit donc les séparer.*

Si on résume ce premier tour de manège, 3 *est en désaccord avec* 0*,* 1 *et* 2*, mais ces trois restent compatibles. On divise donc* N *entre* $N_1 = \{0, 1, 2\}$ *et* $N_2 = \{3\}$*. Puisqu'on a scindé une classe en deux, les résultats de certains "duels" peuvent avoir changé. On refait donc une passe sur* N_1 *:*

- *0 vs. 1*
  - $\delta(0, a) = 1$ *et* $\delta(1, a) = 3$*.*
  - ⇒ 1 *et* 3 *étant maintenant dans des classes différentes, on doit séparer* 0 *et* 1*.*
- *0 vs. 2*
  - $\delta(0, a) = 1$ *et* $\delta(2, a) = 3$*.*
  - ⇒ *Idem, on doit donc séparer* 0 *et* 2*.*
- *1 vs. 2*
  - $\delta(1, a) = 3$ *et* $\delta(2, a) = 3$*.*
  - $\delta(1, b) = 2$ *et* $\delta(2, b) = 1$*.*

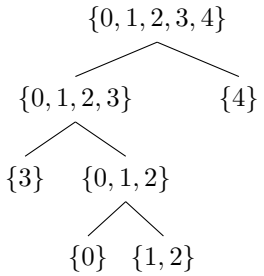
11. En l'occurrence ϵ, mais savoir quel mot exactement n'a pas d'importance.

⇒ *On ne sépare toujours pas* 1 *et* 2.

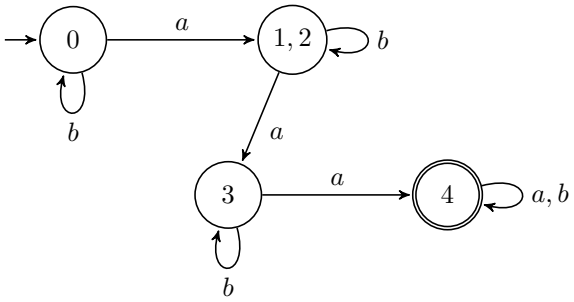
On a donc séparé N_1 *en* $N_2 = \{0\}$ *et* $N_3 = \{1, 2\}$. *Maintenant qu'on a eu une nouvelle séparation, on doit re-tester* 1 *et* 2 *:*

— *1 vs. 2*
  — $\delta(1, a) = 3$ *et* $\delta(2, a) = 3$.
  — $\delta(1, b) = 2$ *et* $\delta(2, b) = 1$.
  ⇒ *On ne sépare toujours pas* 1 *et* 2.

Pour résumer la série des séparations :



On peut maintenant s'amuser à répéter l'expérience autant de fois qu'on veut, mais à partir du moment où il y a eu un tour sans changement, il n'y en aura plus jamais. On accepte alors l'hypothèse selon laquelle 1 *et* 2 *appartiennent à la même classe, ont donc le même rôle et peuvent être fusionnés. On introduit donc un état* $1, 2$. *L'état* 0 *menait initialement avec* a *à l'état* 1*, il pointe donc maintenant vers* $1, 2$. 1 *et* 2 *menaient l'un vers l'autre, ça devient une boucle en* b *sur* $1, 2$. *Enfin,* $1, 2$ *mène avec* a *vers* 3. 1 *et* 2 *n'étaient pas terminaux,* $1, 2$ *ne l'est donc pas non plus :*



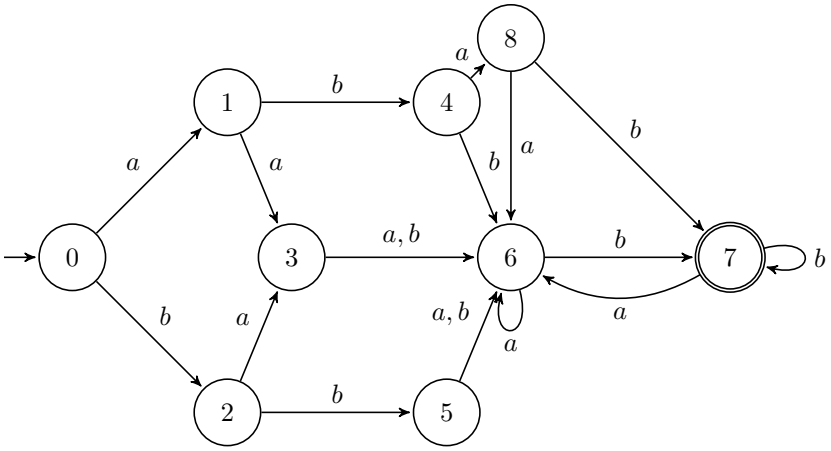
Remarque A la fin de l'exemple, on a justifié de façon un peu rapide et légère l'arrêt de l'algorithme. On n'entrera pas dans les détails ici, mais on donnera l'idée pour que le lecteur ou la lectrice intéressé.e ait matière à réfléchir : l'algorithme présenté ici revient à tester - de façon efficace - les différences d'acceptation entre paires d'états pour les mots de longueur 0, puis 1, puis 2, etc, jusqu'à atteindre un point fixe. Si on n'a pas de changement (ie. de séparation dans une classe) entre les mots de longueur n et $n + 1$, on sait que la stabilité va se propager à l'infini. En effet, soient q_1 et q_2 qui appartiennent à la même classe, et w un mot de longueur $n + 1$. $\delta(q_1, a)$ et $\delta(q_2, a)$ vont dans des états q_1' et q_2' qui appartiennent à la même classe, ce qui veut dire qu'aucun mot de longueur $\leq n + 1$ ne peut les séparer. Alors $\delta^*(q_1, a.w) = \delta^*(q_1', w)$ et $\delta^*(q_2, a.w) = \delta^*(q_2', w)$ sont d'accord, et idem avec b. q_1 et q_2 sont donc d'accord sur les mots de longueur $n + 2$. On peut itérer, ce qui implique qu'ils sont d'accord sur les mots de n'importe

quelle longueur, et sont donc équivalents.

Remarque bis On peut accélérer l'utilisation de cet algorithme en remarquant que la relation "être d'accord" est **transitive** : étant donnée une partition des états, si q_1 et q_2 sont d'accord d'une part, et q_2 et q_3 le sont d'autre part, alors q_1 et q_3 le sont également. De même, si q_1 et q_2 sont d'accord et que q_2 n'est pas d'accord avec q_3, alors ce dernier ne l'est pas avec q_1. On va donc pouvoir s'épargner pas mal de vérifications, comme illustré dans les corrections des deux exercices suivant.

Exercice 3.3.4. *Minimiser (en utilisant l'algorithme) l'automate de l'exemple 3.3.2.*

Exercice 3.3.5. *Minimiser l'automate suivant :*



3.4 Limite de la reconnaissance par automates finis

On s'est jusqu'ici amusé à écrire des automates qui reconnaissent tel ou tel langage, sans jamais faillir. Le théorème suivant va être notre premier échec.

Théorème 2. *Il n'existe pas d'automate fini reconnaissant le langage* $L = \{a^n b^n \mid n \in \mathbb{N}\}$

Démonstration. On va procéder par l'absurde (voir A.1.1), et donc supposer qu'il existe un automate fini A reconnaissant L. Grâce aux algorithmes vus précédemment, on peut supposer que A est déterministe et complet (puisque s'il ne l'est pas, on peut en générer un qui l'est et reconnaît également L).

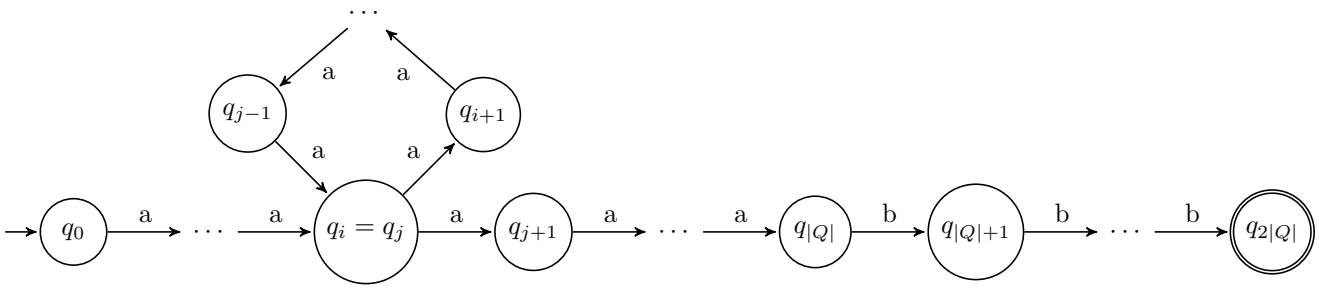
L'ensemble d'états de A, appelé Q, est par définition fini, de cardinal $|Q|$. L contient tous les mots de la forme $a^n b^n$, et notamment $a^{|Q|}b^{|Q|}$. Il existe donc dans A un chemin

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} \dots \xrightarrow{a} q_{|Q|} \xrightarrow{b} q_{|Q|+1} \xrightarrow{b} \dots \xrightarrow{b} q_{2|Q|}$$

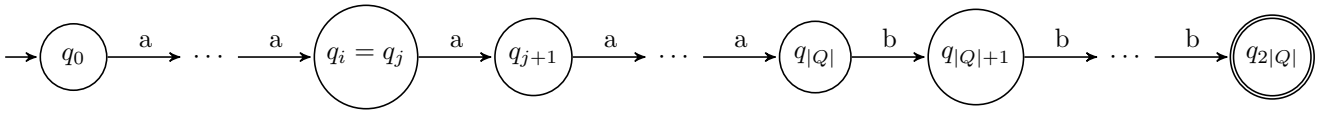
dans A, avec q_0 initial et $q_{2|Q|}$ terminal. On note que, en lisant les a, on parcourt $|Q|+1$ états. D'après le principe des tiroirs (à chaussettes), au moins un état est visité deux fois. On a donc i et j tels que $0 \leq i < j \leq |Q|$ et $q_i = q_j$. A accepte alors le mot $a^{|Q|-(j-i)}b^{|Q|}$ via le chemin

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} ... \xrightarrow{a} q_i \xrightarrow{a} q_{j+1} \xrightarrow{a} ... \xrightarrow{a} q_{|Q|} \xrightarrow{b} q_{|Q|+1} \xrightarrow{b} ... \xrightarrow{b} q_{2|Q|}$$

Autrement dit, on a dans la lecture de $a^{|Q|}b^{|Q|}$ une lecture dans le passage sur les a, et on peut obtenir une nouvelle lecture en retirant le contenu de la boucle (la partie entre i et j). Visuellement, ça ressemble au passage de



à



Or, le mot maintenant lu, $a^{|Q|-(j-i)}b^{|Q|}$, n'appartient pas au langage reconnu par l'automate. On obtient donc un paradoxe, ce qui veut dire que l'automate A n'existait pas en premier lieu. □

Au-delà de la preuve, comment expliquer l'impossibilité pour un automate fini de reconnaître ce langage ? On comparera à l'exemple 3.1.4, où on pouvait raisonner *modulo*, sans retenir toute l'information (le nombre de a et de b lus) mais seulement une abstraction, simplifiée et finie (la parité du nombre de a et de b). Ici, quand on commence a lire des b, on doit avoir enregistré le nombre précis de a lus, pour vérifier qu'on en a bien le même nombre. Or, le nombre de a n'est pas borné, et la seule mémoire disponible dans un automate est les états.

Si on avait le droit à une infinité d'états, on pourrait essayer d'écrire, de manière finie, un automate infini comme celui de la figure 3.8. Le domaine des automates infinis existe bel et bien, mais n'est pas au programme de ce cours. On évoquera, dans la partie 5, une autre approche consistant à ajouter aux automates une deuxième forme de mémoire, potentiellement infinie mais aisément représentable de manière finie.

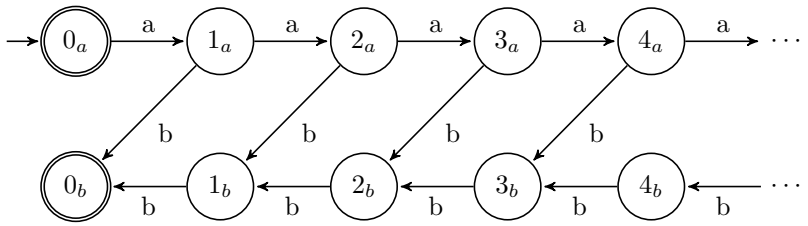


FIGURE 3.8 – Un exemple d'automate infini

3.5 Propriétés de clôture des langages reconnus par automates

On dit qu'un ensemble E est clos sous une opération binaire $\circ$ si, pour tous éléments x et y de E, $x \circ y$ appartient également à E. Dit autrement, E est clos sous $\circ$ ssi. l'application de $\circ$ à ses éléments ne fait pas sortir de l'ensemble. On va regarder ici si les langages reconnaissables par automate fini sont clos sous les opérations sur les langages.

3.5.1 Union

Théorème 3. *Les langages reconnaissables par automate fini sont clos par union, i.e. pour tous langages L_1 et L_2 reconnaissables par automate fini, il existe un automate fini reconnaissant $L_1 \cup L_2$.*

Démonstration. Si L_1 et L_2 sont reconnaissables par automate fini, il existe des automates A_1 et A_2 qui reconnaissent L_1 et L_2, respectivement. Pour plus de généralité, on va supposer A_1 et A_2 non-déterministes. $L_1 \cup L_2$ est reconnu par "A_1 et A_2 posés à côté l'un de l'autre". Plus formellement, soient $A_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ et $A_1 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2 \rangle$, alors $L_1 \cup L_2$ est reconnu par $\langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta', I_1 \cup I_2, F_1 \cup F_2 \rangle$ avec

$$\begin{cases} \delta'(q,a) = \delta_1(q,a) & \text{si } q \in Q_1, \\ \delta'(q,a) = \delta_2(q,a) & \text{si } q \in Q_2 \end{cases}$$

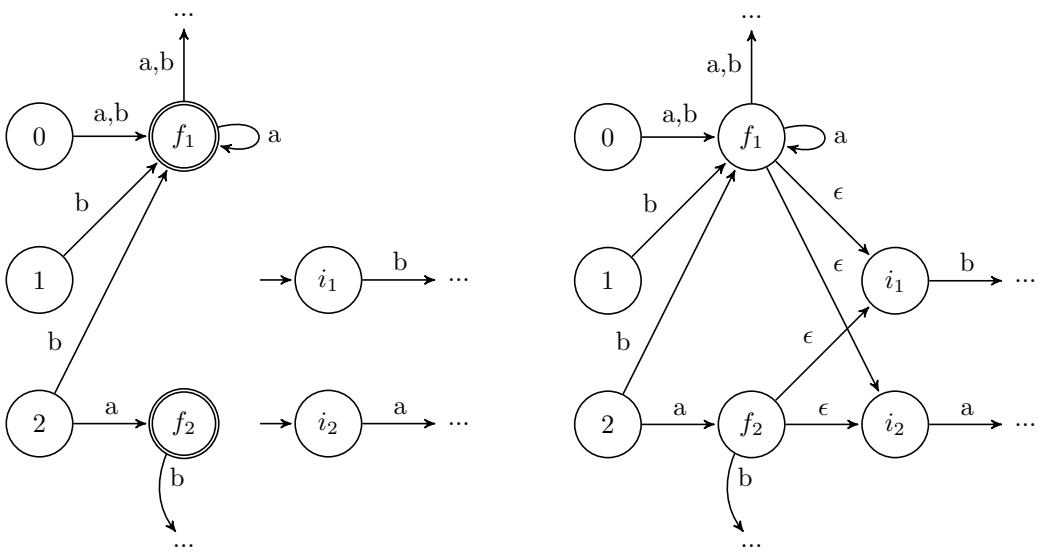
□

C'est exactement l'astuce qui a été utilisée dans l'exemple 3.2.2, où on a effectivement "mis ensemble" deux automates.

3.5.2 Concaténation

Théorème 4. *Les langages reconnaissables par automate fini sont clos par concaténation, i.e. pour tous langages L_1 et L_2 reconnaissables par automate fini, il existe un automate fini reconnaissant $L_1.L_2$.*

Démonstration. Si L_1 et L_2 sont reconnaissables par automate fini, il existe des AF A_1 et A_2 qui reconnaissent L_1 et L_2, respectivement. On veut reconnaître $L_1.L_2 = \{u.v \mid u \in L_1 \ \wedge \ v \in L_2\}$, l'astuce va être de mettre A_1 et A_2 l'un après l'autre en rajoutant des ϵ-transitions pour que, à chaque fois qu'on finit de lire un mot u accepté par A_1, au lieu d'accepter, on puisse aller vers un état initial de A_2 en lire un deuxième v. Dualement au fait qu'on ne veuille plus accepter dès la fin d'une lecture en A_1, on ne veut pas pouvoir commencer en A_2. La figure 3.9 illustre schématiquement cette transformation sur un exemple.



(a) Deux automates voisins isolés ... (b) maintenant reliés

FIGURE 3.9 – Concaténation d'automates

Soit un mot $w = u.v$ avec $u \in L_1$ et $v \in L_2$. u étant accepté par A_1, sa lecture en partant d'un état initial nous amène en f_1 ou f_2. Dans la version transformée, u seul n'est plus accepté, mais les ϵ-transitions partant de f_1 et f_2 nous permettent de nous "téléporter" en i_1 ou i_2. Or, v est accepté par A_2, ce qui veut dire qu'il va nous envoyer de l'un ou l'autre à un état terminal de A_2.

Plus formellement, soient $A_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ et $A_1 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2 \rangle$, alors $L_1.L_2$ est reconnu par $\langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta', I_1, F_2 \rangle$ avec

$$\begin{cases} \delta'(q,a) = \delta_1(q,a) & \text{si } q \in Q_1 \\ \delta'(q,\epsilon) = \delta_1(q,a) & \text{si } q \in Q_1 \backslash F_1, \\ \delta'(q,\epsilon) = \delta_1(q,a) \cup I_2 & \text{si } q \in F_1, \\ \delta'(q,a) = \delta_2(q,a) & \text{si } q \in Q_2, \\ \delta'(q,\epsilon) = \delta_2(q,\epsilon) & \text{si } q \in Q_2 \end{cases}$$

Ainsi, on garde toutes les transitions autrefois présentes, et on ajoute (troisème cas) des ϵ-transitions des états terminaux de A_1 vers les initiaux de A_2.

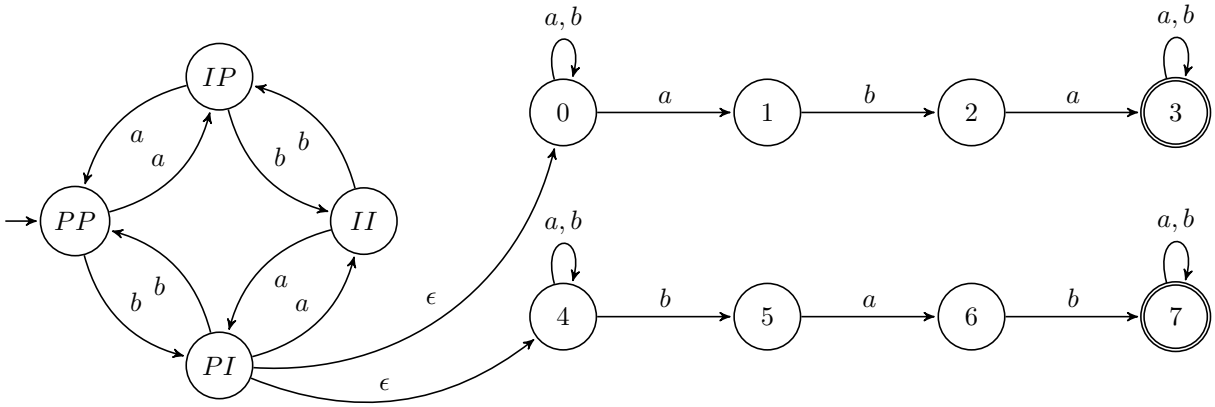
Si A_1 accepte le mot vide, on veut pouvoir considérer cette possibilité et commencer la lecture en A_2 directement. Or, c'est déjà pris en compte par les nouvelles transitions : A_1 accepte le mot vide ssi. il a un état initial et terminal. Or, la définition de
delta′ ajoute une ϵ-transition de cet état vers tous les initiaux de A_2. On peut donc, en pratique, faire lire tout mot donné intégralement par A_2.

Pas besoin non plus de faire attention pour les états terminaux : si A_2 accepte le mot vide, au moins un de ses états initiaux est terminal, ce qui veut dire que les transitions qu'on ajoute

permettront d'accepter un mot de L_1. □

Remarque On a dit en 3.2.1 qu'on ignorerait dans ce document les ϵ-transitions, à l'exception de la sous-section actuelle et la suivante (3.5.3), d'où la prise en compte des ϵ dans la définition de δ'.

Exemple 3.5.1. *Soient $L_1 = \{w \mid |w|_a$ pair et $|w|_b$ impair$\}$ et $L_2 = \Sigma^*(aba + bab)\Sigma^*$. On peut reconnaître $L_1.L_2$ en combinant les automates des exemples 3.1.4 et 3.2.2. L'état anciennement terminal PI permet maintenant d'accéder gratuitement à 0 et 4, qui deviennent non-initiaux. On obtient au final :*



3.5.3 Itération

Théorème 5. *Les langages reconnaissables par automate fini sont clos par concaténation, i.e. pour tout langage L reconnaissable par automate fini, il existe un automate fini reconnaissant L^*.*

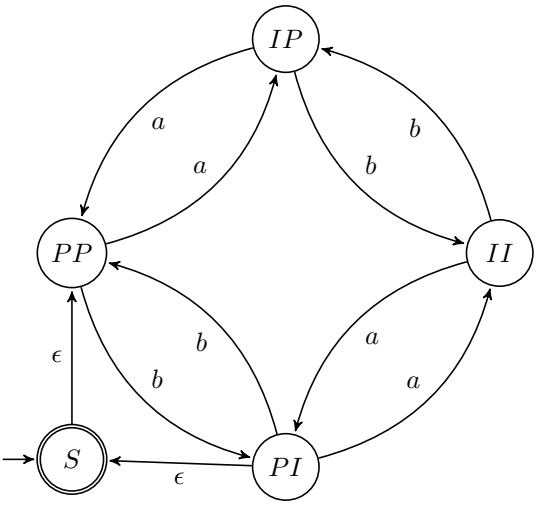
Démonstration. On veut reconnaître des mots de la forme $u_1.u_2...u_n$ avec $u_i \in L$ pour tout i. On ajoute un état spécial, S, qui sera le seul initial et terminal, et qui sera le début et la fin de tout tour de manège. La construction de l'automate ressemble ensuite à celle de la concaténation, en ce qu'on rajoute des ϵ-transitions qui, quand on est dans un état (anciennement) terminal, nous permettent de revenir à S. Dualement, on ajoute des ϵ-transitions allant de S à tout état (anciennement) initial pour pouvoir commencer chaque tour.

Plus formellement, soient $A = \langle Q, \Sigma, \delta, I, F\rangle$, alors L^* est reconnu par $\langle Q \cup \{S\}, \Sigma, \delta', \{S\}, \{S\}\rangle$ avec

$$\begin{cases} \delta'(q,a) = \delta(q,a) & \text{si } q \in Q, \\ \delta'(q,\epsilon) = \delta(q,\epsilon) & \text{si } q \in Q \backslash F, \\ \delta'(q,\epsilon) = \delta(q,\epsilon) \cup \{S\} & \text{si } q \in F, \\ \delta'(S,a) = \emptyset, \\ \delta'(S,\epsilon) = I \end{cases}$$

□

Exemple 3.5.2. *Soit $L = \{w \in \Sigma^* \mid |w|_a$ pair et $|w|_b$ impair$\}$. L^* est reconnu par l'automate suivant :*



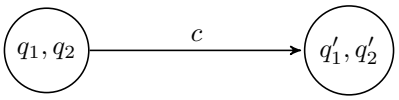
Exercice 3.5.1. *Est-il nécessaire dans la transformation de rendre non-terminaux les états qui l'étaient dans l'automate original ?*

Exercice 3.5.2. *Pourquoi était-il indispensable d'introduire un nouvel état dans la transformation ?*

3.5.4 Intersection

Théorème 6. *Les langages reconnaissables par automate fini sont clos par intersection, i.e. pour tous langages L_1 and L_2 reconnaissables par automate fini, il existe un automate fini reconnaissant $L_1 \cap L_2$.*

Démonstration. Soient A_1 et A_2 des automates reconnaissant L_1 et L_2. Comme dans l'algorithme de déterminisation, on va utiliser un automate pour en simuler un autre, ou en l'occurrence deux, en utilisant comme états des couples d'états de A_1 / A_2. Si par exemple on a dans $q_1 \xrightarrow{c} q_1'$ dans A_1 et $q_2 \xrightarrow{c} q_2'$ dans A_2, alors on aura dans l'automate de $L_1 \cap L_2$



Ce faisant, la lecture d'un mot exécute celles du même mot à la fois dans A_1 (parties gauches des différents états) et dans A_2 (parties droites). On veut commencer au début des deux automates, les états initiaux seront donc l'ensemble des i_1, i_2 tels que i_1 est initial dans A_1 et i_2 dans A_2. On ne veut accepter un mot que s'il appartient aux deux langages, cad. ssi. il serait accepté par les deux langages. Les états terminaux sont donc ceux de la forme f_1, f_2, où f_1 et f_2 sont tous les deux terminaux dans leurs automates respectifs.

Plus formellement, soient $A_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ et $A_1 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2 \rangle$, alors $L_1 \cap L_2$ est reconnu par $\langle Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta', I_1 \times I_2, F_1 \times F_2 \rangle$ avec $\delta(\langle q_1, q_2 \rangle, a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$.

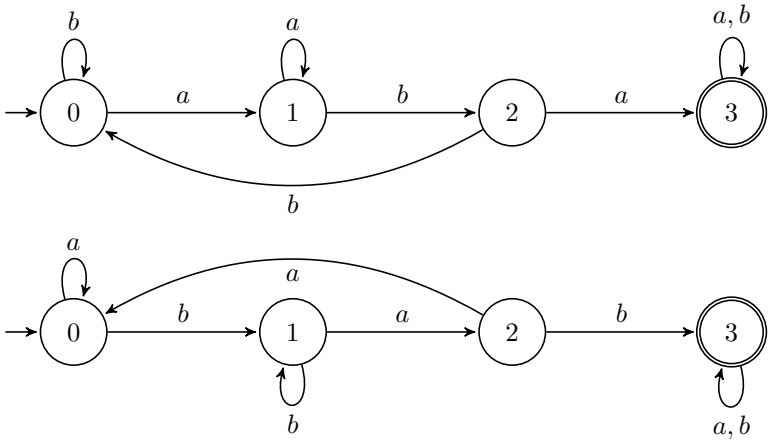
Par coquetterie et amour inconditionnel des homomorphismes, on mentionnera la jolie définition de δ' dans le cas déterministe :

$$\delta(\langle q_1, q_2 \rangle, a) = \langle \delta_1(q_1, a), \delta_2(q_2, a) \rangle$$

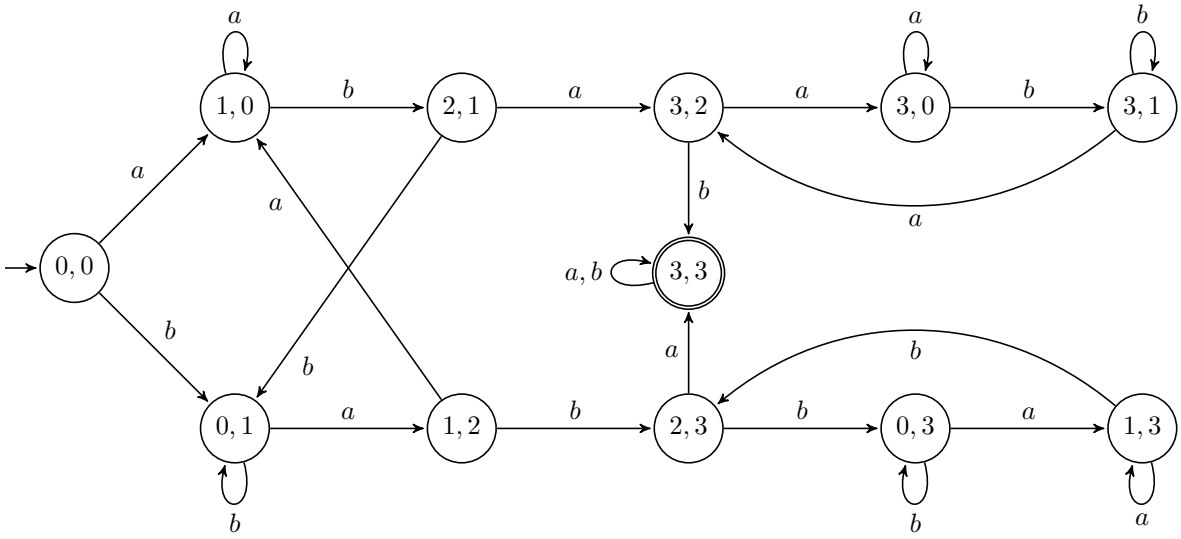
□

Remarque L'intersection de deux automates est déterministe ssi. les deux automates sont déterministes.

Exemple 3.5.3. *On veut reconnaître le langage des mots qui contiennent les facteurs aba et bab. Les facteurs se chevauchant, écrire directement un tel automate n'est pas totalement évident. On va utiliser l'algorithme d'intersection, dans sa version déterministe ou non. On choisit ici la première option. On commence donc en rappelant les AFD reconnaissant les deux langages à mêler :*



On part de l'état $0, 0$. *Vu que* a *nous emmène de* 0 *en* 1 *dans le premier automate et de* 0 *en* 0 *dans le second,* $0, 0$ *va en* $1, 0$ *et* $1, 0$ *en lisant* a, *et ainsi de suite. On obtient au final l'automate suivant :*



Exercice 3.5.3. *Minimiser l'automate de l'exemple 3.5.3.*

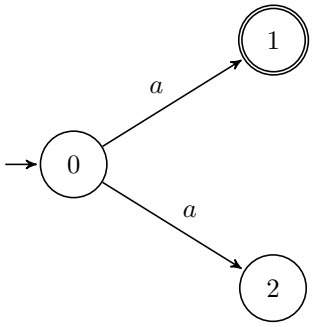
Exercice 3.5.4. *Appliquer l'algorithme d'intersection sur les versions non-déterministes des automates de l'exemple 3.5.3 (cf. l'exemple 3.2.2).*

3.5.5 Complémentaire

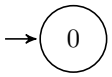
Théorème 7. *Les langages reconnaissables par automate fini sont clos par complémentaire, i.e. pour tout langage L reconnaissable par automate fini, il existe un automate fini reconnaissant $\overline{L}$, cad. le langage des mots qui n'appartiennent pas à L.*

Démonstration. Puisqu'on veut reconnaître ce qui ne l'était pas, et ne plus reconnaître ce qui l'était, une idée assez naturelle est d'inverser les états terminaux et non-terminaux. Il y a cependant deux subtilités.

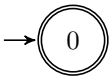
Il faut d'abord que l'état soit déterministe. En effet, si on a un AFND et un mot ayant dans l'automate deux chemins, un menant à un état terminal et un menant à un état non-terminal, alors le mot sera accepté avant et après inversion. On peut observer ce phénomène concrètement avec le mot a et l'automate



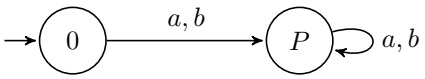
De plus, l'automate doit être complet. En effet, un chemin non-existant reste non-existant malgré une inversion des terminaux et non-terminaux. Par exemple le mot a n'est pas reconnu par l'automate



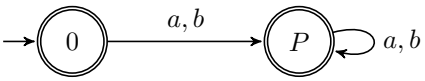
ni par l'automate



Si on complète le premier automate, on obtient



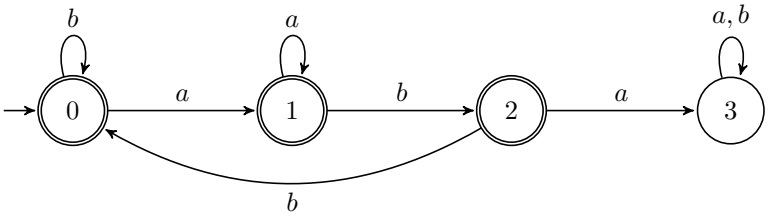
qui ne reconnait toujours aucun mot. En calculant son complémentaire, on obtient



qui accepte tous les mots.

□

Exemple 3.5.4. *On veut un automate pour les mots qui ne contiennent pas le facteur aba. En reprenant un exemple qu'on a déjà trop vu, on obtient*



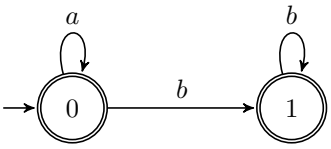
Exercice 3.5.5. *Donner un automate des mots ne terminant pas par aba.*

3.5.6 Conséquences pour les langages non-reconnus

On a vu que les langages reconnaissables par automate fini pouvaient se construire en combinant d'autres langages reconnaissables par automate fini, ce qui nous permet de montrer facilement que tout un tas de langages sont eux-mêmes reconnaissables par un automate fini. Mais ces propriétés de clôture permettent aussi, assez ironiquement, de montrer facilement que pas mal de langages ne le sont pas.

Théorème 8. *Le langage $L = \{w \mid |w|_a = |w|_b\}$, cad. l'ensemble des mots contenant autant de a que de b, n'est pas reconnaissable par automate fini.*

Démonstration. On pourrait essayer d'adapter la preuve du théorème 2, mais ça serait beaucoup plus désagréable que d'utiliser la clôture par intersection. En effet, le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ peut être défini comme l'intersection de a^*b^* et L. Or, a^*b^* est reconnu par l'automate suivant :



Si L était reconnu par un automate fini, $a^*b^* \cap L = \{a^n b^n \mid n \in \mathbb{N}\}$ le serait également, ce qui n'est pas le cas (cf. théorème 2). On en déduit donc que L n'est pas reconnaissable par automate.

□

Exercice 3.5.6. *Montrer que le langage $\{w \in \{a, b, c\}^* \mid |w|_a = |w|_b\}$ n'est pas reconnaissable par automate.*

Chapitre 4

Théorème de Kleene

On a pour l'instant utilisé les expressions rationnelles pour décrire les langages reconnus par des automates, et utilisé des automates pour implémenter des expressions rationnelles. On a cependant vu que l'expressivité des automates était limitée, en ce qu'il existe de nombreux langages qu'ils ne peuvent reconnaître. On peut donc légitimement se demander si les expressions rationnelles sont plus ou moins expressives que les automates finis.

Le mathématicien Stephen C. Kleene (1909 - 1994) répond à cette question via le théorème qui porte son nom :

Théorème 9. ***(Théorème de Kleene)*** *Les langages représentable par expression rationnelle sont exactement ceux reconnus par automate fini.*

L'égalité entre deux ensembles A et B équivaut à la double inclusion entre les mêmes ensembles, cad. $A \subseteq B \wedge B \subseteq A$. On va donc chercher une preuve que chaque langage représentable par *regex* est reconnaissable par automate fini, et inversement. Ces deux sous-théorèmes admettent aujourd'hui de nombreuses preuves, notamment constructives, cad. sous la forme d'algorithmes qui réalisent la transformation d'une *regex* en automate, et inversement. Cette section présente plusieurs de ces algorithmes.

4.1 Des expressions rationnelles aux automates

On veut prouver ici le théorème suivant :

Théorème 10. *Les langages représentables par expression rationnelle sont également reconnaissables par automate fini.*

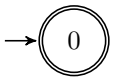
On en présente ici deux preuves différentes.

4.1.1 Traduction récursive

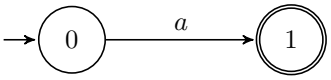
Démonstration. On va procéder par induction structurelle sur les expressions rationnelles. On en rappelle d'abord la définition récursive :

$$
\begin{array}{rcl}
e & ::= & \epsilon \\
& | & a \in \Sigma \\
& | & e_1.e_2 \\
& | & e_1 + e_2 \\
& | & e_1^*
\end{array}
$$

L'expression ϵ est clairement reconnu par l'automate



L'expression a quant à elle est reconnue par



On peut donc traduire en automates les cas de base des expressions rationnelles. Quant aux cas récursifs, on les a en fait déjà traités en 3.5 : pour construire un automate reconnaissant $e_1 + e_2$, on construit des automates A_1 et A_2 reconnaissant e_1 et e_2, et on applique la construction de l'union. On procède de même avec la concaténation pour $e_1 e_2$ et l'itération pour e^*. □

Remarque Les constructions vues pour les propriétés de clôture ne sont pas optimales. L'algorithme de Thomson utilise les mêmes idées mais s'assure qu'il n'y ait pas plus d'un état initial et un état terminal (différents) tout au long de la construction de l'automate, ce qui permet de borner plus finement sa taille, et de faciliter la concaténation (on fusionne l'état terminal gauche et l'état initial droit).

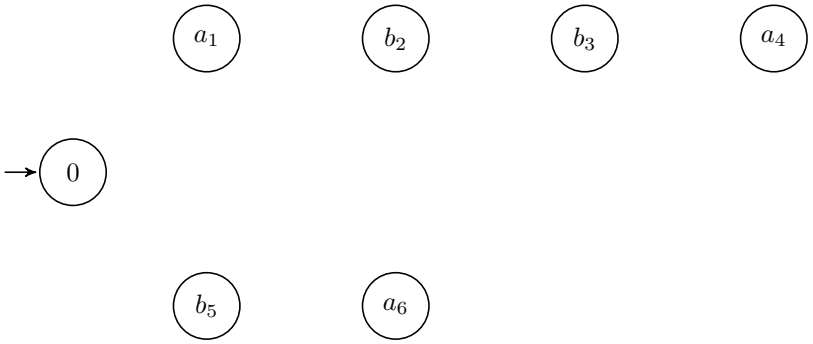
4.1.2 Traduction linéaire

On propose un autre algorithme, qui sépare la construction des états de celles des transitions : l'**algorithme de Glushkov**.

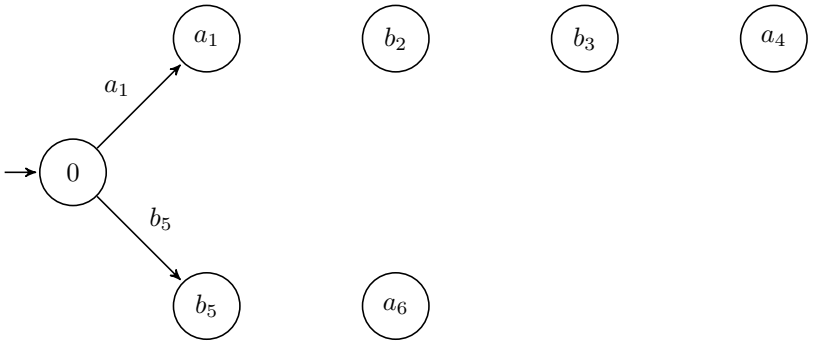
Exemple 4.1.1. *Soit $e = (abb^*a + (ba)^*)^*$. On commence par distinguer toutes les lettres de l'expression, par exemple en les indexant par leur position. On obtient donc*

$$(a_1 b_2 b_3^* a_4 + (b_5 a_6)^*)^*$$

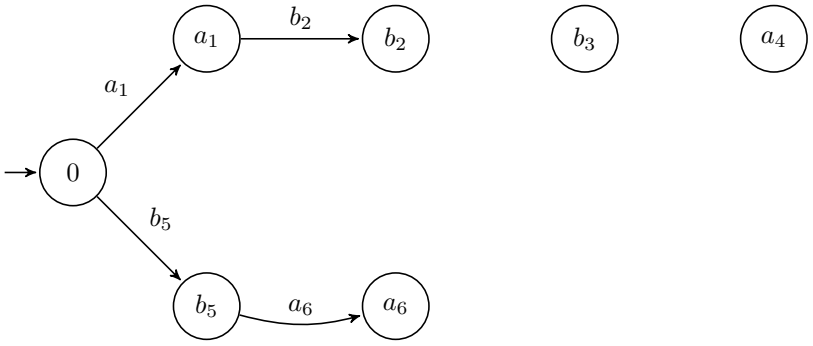
On crée ensuite un état pour chaque lettre, ainsi qu'un état initial 0 :



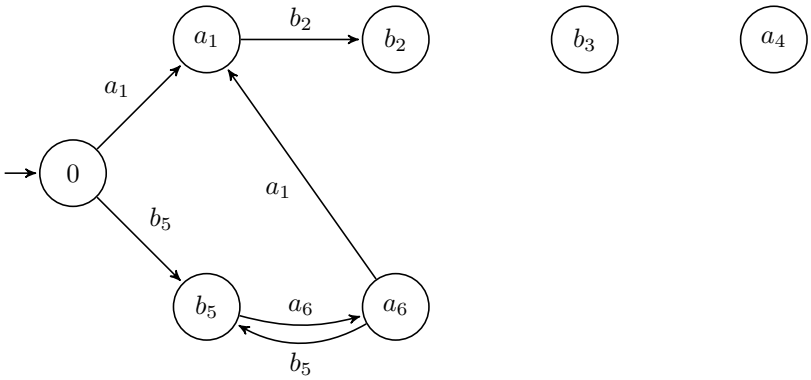
Le jeu va être de représenter, avec les transitions, les "promenades" dans l'expression. Toute mot capturé par l'expression commence par a_1 ou b_5. On ajoute donc ces transitions :



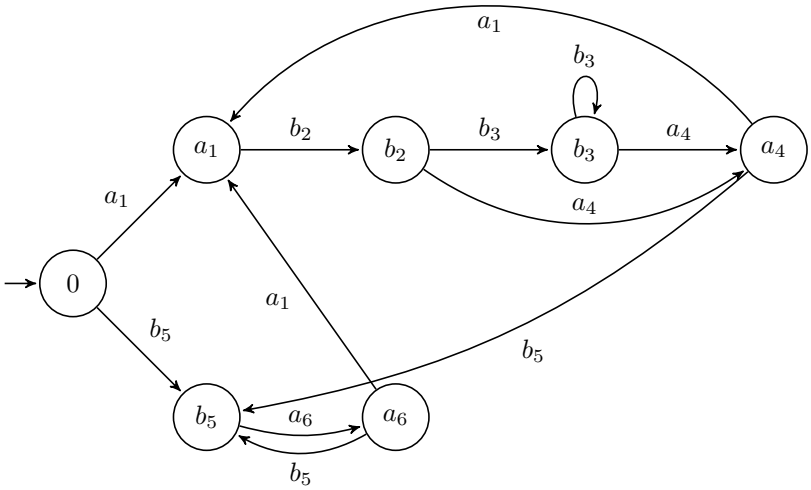
Quand on a lu un a_1, on est obligé de lire un b_2. De même, un b_5 est forcément suivi d'un a_6 :



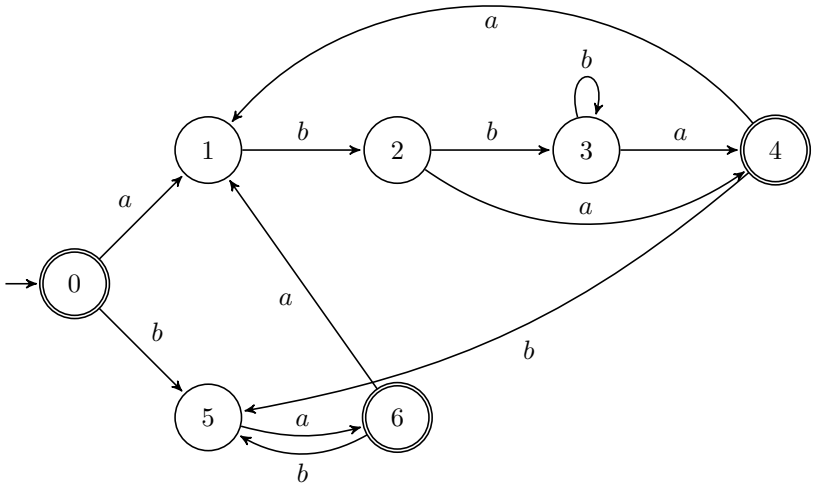
Après la lecture d'un a_6, on peut recommencer la boucle $(b_5a_6)^*$, auquel cas on lit un b_5, ou recommencer la boucle entière $(a_1b_2b_3^*a_4 + (b_5a_6)^*)^*$, en lisant un a_1 ou un b_5 :



Après la lecture d'un b_2, on peut lire un b_3, mais aussi sauter b_3^* et aller directement lire un a_4. En b_3, on peut boucler ou passer en a_4. En a_4, comme en a_6, on peut lire a_1 ou b_5. On a donc :



On peut arrêter de boucler quand on vient de lire un a_4 ou un a_6, ce qui veut dire que les états correspondant vont être terminaux. De plus, l'expression rationnelle contient le mot vide, ce qui veut dire que l'état initial doit être terminal. Il ne nous reste plus qu'à "déspécialiser" les lettres, et éventuellement renommer les états :



On a présenté l'algorithme *via* un exemple, mais il repose bien évidemment sur des définitions formelles, procédant par induction sur l'expression. Une fois qu'on a spécialisé les lettres et crée les états comme dans l'exemple, on veut d'abord savoir si l'expression rationnelle donnée accepte ou non le mot vide. C'est à cette question que répond la fonction suivante :

$$\begin{cases} E(\epsilon) = \top \\ E(a) = \bot \\ E(e_1 + e_2) = E(e_1) \vee E(e_2) \\ E(e_1.e_2) = E(e_1) \wedge E(e_2) \\ E(e^*) = \top \end{cases}$$

L'état initial est terminal ssi. $E(e) = \top$. Pour les premières transitions, qui partiront de l'état initial 0, on a besoin de déterminer quelles lettres peuvent commencer les mots reconnus par l'expression, ce que fait la fonction suivante :

$$\begin{cases} D(\epsilon) = \emptyset \\ D(a) = \{a\} \\ D(e_1 + e_2) = D(e_1) \cup D(e_2) \\ D(e_1.e_2) = D(e_1) & \text{Si } e_1 \text{ n'accepte pas le mot vide, cad. si } \neg E(e_1) \\ D(e_1.e_2) = D(e_1) \cup D(e_2) & \text{sinon} \\ D(e^*) = D(e) \end{cases}$$

On ajoute une transition $0 \xrightarrow{q} q$ pour tout $q \in D(e)$. De même, on a besoin de savoir quelles lettres peuvent finir les mots du langage dénoté par une expression :

$$\begin{cases} F(\epsilon) = \emptyset \\ F(a) = \{a\} \\ F(e_1 + e_2) = F(e_1) \cup F(e_2) \\ F(e_1.e_2) = F(e_2) & \text{Si } e_2 \text{ n'accepte pas le mot vide, cad. si} \neg E(e_2) \\ fF(e_1.e_2) = F(e_1) \cup F(e_2) & \text{sinon} \\ F(e^*) = F(e) \end{cases}$$

Tout les états de F sont terminaux. En plus des premières transitions, on veut bien sûr les autres. Pour ça, on calcule l'ensemble des lettres qui peuvent se suivre dans le langage dénoté par l'expression :

$$\begin{cases} P(\epsilon) = \emptyset \\ P(a) = \emptyset \\ P(e_1 + e_2) = P(e_1) \cup P(e_2) \\ P(e_1.e_2) = P(e_1) \cup P(e_2) \cup F(e_1).D(e_2) \\ P(e^*) = P(e) \cup F(e).D(e) \end{cases}$$

Toute paire $\langle q_1, q_2 \rangle$ appartenant à $P(e)$ génère donc une transition $q_1 \xrightarrow{q_2} q_2$ dans l'automate.

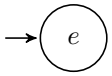
Exercice 4.1.1. *Utiliser l'algorithme de Glushkov pour traduire en automate l'expression* $e = (bb)^*(b(a+b)^*)^*$

4.1.3 Méthode des résiduels

Janusz A. Brzozowski introduit en 1964 la notion de dérivée partielle d'expression rationnelle, parfois appelée **dérivée de Brzozowski**, qu'il utilise dans un algorithme traduisant une regex en automate fini, déterministe et complet. En 1996, Valentin Antimirov propose une variation de la dérivation et de l'algorithme, qui produit un automate potentiellement non-déterministe mais minimal. On donnera ici une version quelque peu édulcorée de cette méthode générale via un exemple, pour en faire passer l'idée, puis on en fournira les bases techniques.

Exemple 4.1.2. *On veut obtenir un automate reconnaissant le langage dénoté par l'expression* $c(ab)^*ac + aac(ab)^*$. *Pour ça, on va utiliser des états dont les noms sont, sous forme d'expression rationnelle, le langage qu'il nous reste à lire une fois qu'on les a atteints. Ce "langage qu'il nous reste à lire" est ce qu'on appelle un* ***résiduel****.*

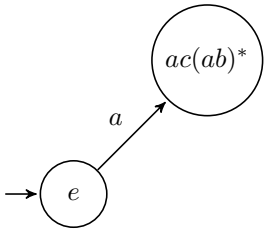
Au tout début de la lecture d'un mot, l'expression qu'il nous reste à reconnaître est bien évidemment $c(ab)^*ac + aac(ab)^*$ *(qu'on écrira* e*). On a donc notre état initial :*



Il faut ensuite déterminer ce qu'il nous resterait à lire après un a pour être dans l'expression e. On note ce langage $a^{-1}e = a^{-1}(c(ab)^*ac + aac(ab)^*)$. Puisque e est l'ensemble des mots de la forme $c(ab)^*ac$ ou $aac(ab)^*$, $a^{-1}e$ est l'ensemble des mots de la forme $a^{-1}c(ab)^*ac$ ou $a^{-1}aac(ab)^*$.

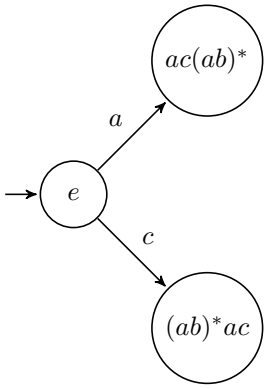
La notation $a^{-1}c(ab)^*ac$ dénote l'ensemble des mots u tels que au appartienne au langage $c(ab)^*ac$. On voit bien qu'il y a une incompatibilité au niveau de la première lettre, ce langage est donc vide.

En revanche, la notation $a^{-1}aac(ab)^*$ n'est pas vide. Il s'agit en effet de l'ensemble des mots u tels que au soit de la forme $aac(ab)^*$, ensemble dénoté par $ac(ab)^*$. On intègre cette donnée à l'automate, en ajoutant un état et une transition disant que, quand on doit reconnaître e et qu'on lit un a, il nous reste à lire un mot de la forme $ac(ab)^*$:



De même, on calcule $b^{-1}e = b^{-1}(c(ab)^*ac + aac(ab)^*) = b^{-1}(c(ab)^*ac) + b^{-1}(aac(ab)^*)$. On voit que le langage est vide, puisqu'on cherche des mots qui commencent à la fois par b et soit c soit a. On n'ajoute donc pas d'état ou de transition.

On calcule aussi $c^{-1}e = c^{-1}(c(ab)^*ac + aac(ab)^*) = c^{-1}(c(ab)^*ac) + c^{-1}(aac(ab)^*)$. La partie droite du résiduel est vide, mais pas la partie gauche. On a en effet $c^{-1}c(ab)^*ac + c^{-1}aac(ab)^* = (ab)^*ac$. On ajoute donc l'état et la transition correspondant :



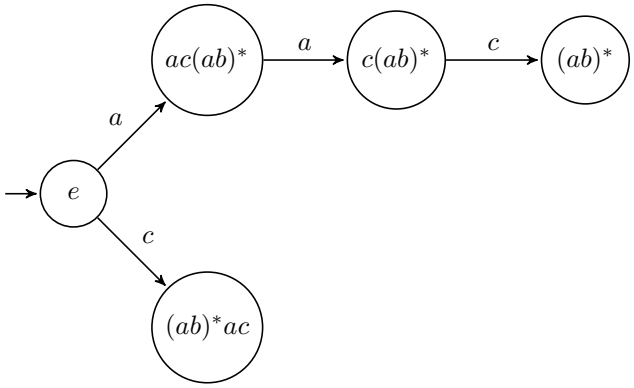
On continue sur la partie haute de l'automate. On a

— $a^{-1}ac(ab)^* = c(ab)^*$
— $b^{-1}ac(ab)^* = c^{-1}ac(ab)^* = \emptyset$

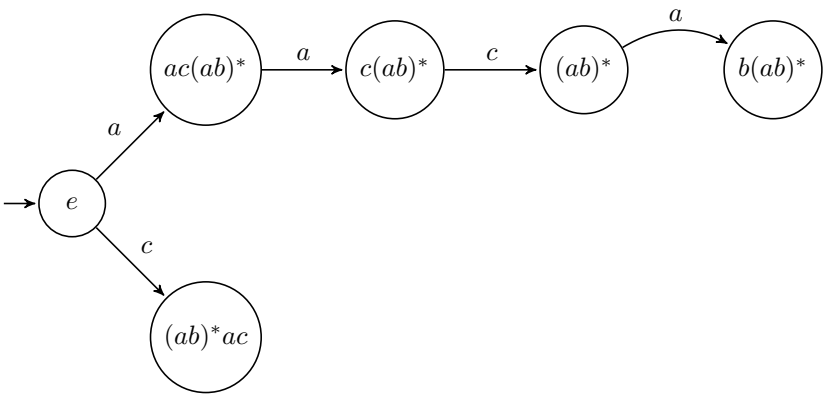
puis

— $c^{-1}c(ab)^* = (ab)^*$
— $a^{-1}c(ab)^* = b^{-1}c(ab)^* = \emptyset$

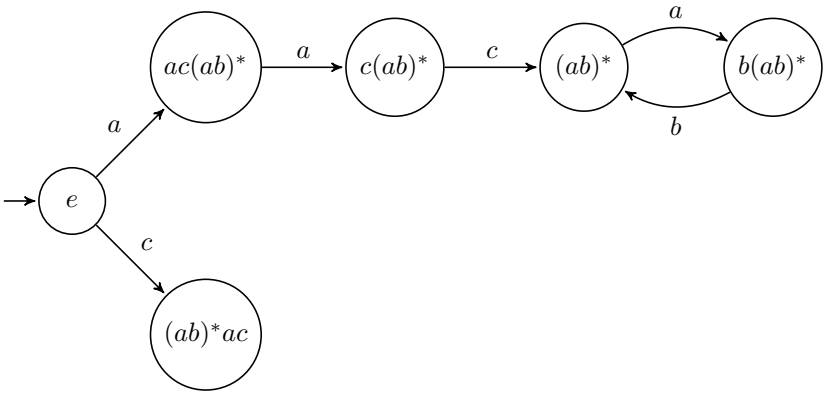
On met à jour l'automate :



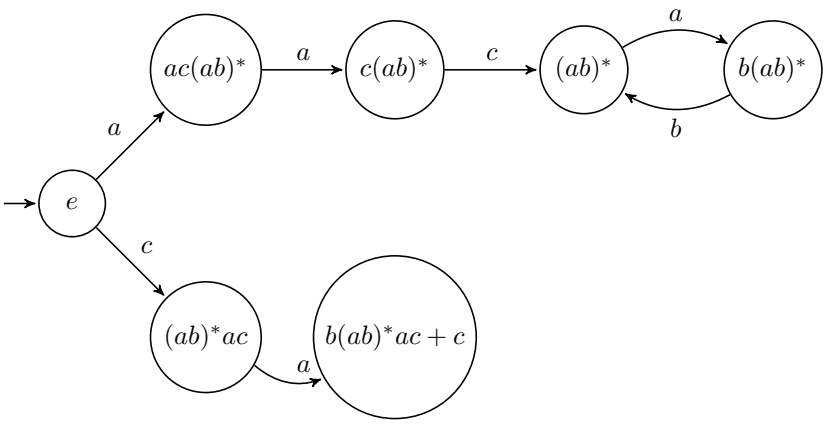
On s'intéresse ensuite à $(ab)^*$. Si un mot de $(ab)^*$ commence par a, c'est que l'étoile a été utilisée au moins une fois. Dans ce cas, l'expression "visée" est en fait $ab(ab)^*$. Le résiduel $a^{-1}(ab)^*$ est donc $a^{-1}ab(ab)^* = b(ab)^*$. Par le même raisonnement, $b^{-1}(ab)^* = c^{-1}(ab)^* = \emptyset$:



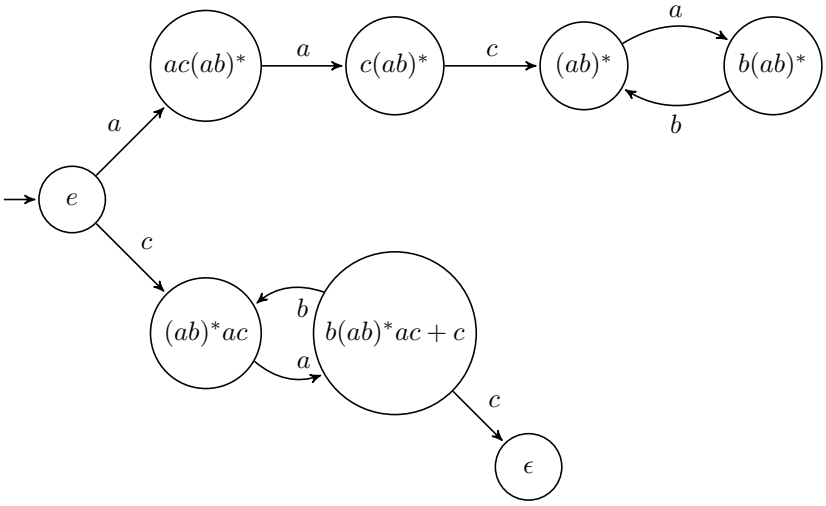
Le seul résiduel non-vide de $b(ab)^*$ *est* $b^{-1}b(ab)^* = (ab)^*$*. On revient donc sur l'état précédent :*



On revient maintenant sur la partie basse de l'automate, en particulier l'état $(ab)^*ac$*. On peut unfolder manuellement l'étoile, en séparant les cas où elle est instanciée zéro ou au moins une fois, ce qui donne l'expression* $ab(ab)^*ac + ac$*. On peut maintenant calculer le résiduel* $a^{-1}(ab)^*ac = a^{-1}(ab(ab)^*ac + ac) = b(ab)^*ac + c$*. Les résiduels en* b *et* c *sont par contre vides :*



En calculant les derniers résiduels, on arrive à



Il ne nous manque plus que les états terminaux. On devrait pouvoir arrêter notre lecture ssi. le langage qu'il nous reste à reconnaître contient le mot vide. Les états terminaux sont donc ceux dont l'expression rationnelle dénote un langage qui contient ϵ. Cette condition peut se calculer en utilisant la fonction E donnée dans en 4.1.2. Au final, l'automate est donc

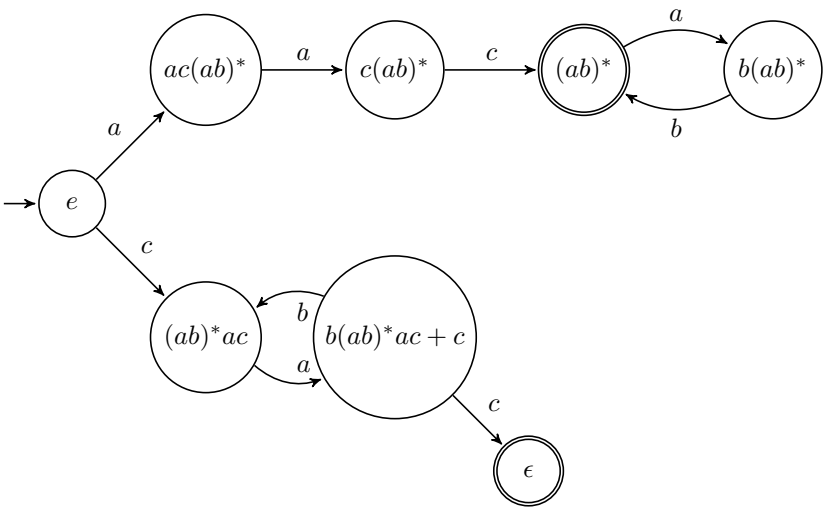


FIGURE 4.1 – Automate reconnaissant $c(ab)^*ac + aac(ab)^*$

On a fait l'exemple précédent *avec les mains*, mais la dérivée est bien sûr formellement définie comme suit :

$$\begin{cases} a^{-1}\epsilon = \emptyset \\ a^{-1}a = \epsilon \\ a^{-1}b = \emptyset \\ a^{-1}(e_1 + e_2) = a^{-1}e_1 + a^{-1}e_2 \\ a^{-1}(e_1.e_2) = (a^{-1}e_1)e_2 & \text{Si } e_1 \text{ n'accepte pas le mot vide, cad. si } \neg E(e_1) \\ a^{-1}(e_1.e_2) = (a^{-1}e_1)e_2 + a^{-1}e_2 & \text{sinon} \\ a^{-1}e^* = (a^{-1}e)e^* \end{cases}$$

Exemple 4.1.3. *En appliquant ces calculs à l'expression $c(ab)^*ac + aac(ab)^*$, on retrouve la construction de notre automate :*

- $a^{-1}(c(ab)^*ac + aac(ab)^*) = a^{-1}c(ab)^*ac + a^{-1}aac(ab)^* = a^{-1}aac(ab)^* = ac(ab)^*$
  - $a^{-1}ac(ab)^* = c(ab)^*$
    - $a^{-1}c(ab)^* = b^{-1}c(ab)^* = \emptyset$
    - $c^{-1}c(ab)^* = (ab)^*$
      - $a^{-1}(ab)^* = b(ab)^*$

        $a^{-1}b(ab)^* = c^{-1}b(ab)^* = \emptyset$

        $b^{-1}b(ab)^* = (ab)^*$
      - $b^{-1}(ab)^* = c^{-1}(ab)^* = \emptyset$
  - $b^{-1}ac(ab)^* = c^{-1}ac(ab)^* = \emptyset$
- $b^{-1}(c(ab)^*ac + aac(ab)^*) = \emptyset$
- $c^{-1}(c(ab)^*ac + aac(ab)^*) = c^{-1}c(ab)^*ac + c^{-1}aac(ab)^* = c^{-1}c(ab)^*ac = (ab)^*ac$
  - $a^{-1}(ab)^*ac = b(ab)^*ac + c$
    - $a^{-1}(b(ab)^*ac + c) = \emptyset$
    - $b^{-1}(b(ab)^*ac + c) = (ab)^*ac$

— $c^{-1}(b(ab)^*ac + c) = \epsilon$
— $b^{-1}(ab)^*ac = c^{-1}(ab)^*ac = \emptyset$

Une grande difficulté, qu'on a évitée dans l'exemple ci-dessus, est l'identification d'expressions rationnelles équivalentes. En effet, si on applique la méthode des résiduels à l'expression $(aa)^* + aa$, la dérivée par aa va nous emmener sur $(aa)^* + \epsilon$, qui est une expression syntaxiquement différente, mais sémantiquement identique, en ce qu'elle dénote le même langage. C'est sur la gestion de ce problème que différent les propositions de Brzozowski et Antimirov, différences qui ne sont pas étudiées ici.

A supposer qu'on arrive à éviter ce genre de problème, la méthode présentée ici produit systématiquement un automate déterministe et minimal reconnaissant le même langage que l'expression rationnelle donnée en argument.

Remarque On a présenté dans cette section différents algorithmes comme des preuves de la "traductabilité" des expressions rationnelles en automates finis. Si on était vraiment rigoureux, les algorithmes ne suffiraient pas, il faudrait aussi prouver 1) qu'ils terminent sur toute entrée 2) qu'ils produisent bien un automate fini acceptant le langage décrit par l'expression donnée. Ces preuves sont autrement plus complexes que l'écriture des algorithmes [1] et vont au-delà du programme du cours, mais il est toujours bon de garder son esprit critique face à ces choses. Cette remarque s'applique bien évidemment également à l'algorithme présenté dans les pages qui suivent pour la traduction inverse.

4.2 Des automates aux expressions rationnelles - algorithme de McNaughton et Yamada

On veut maintenant prouver le théorème suivant :

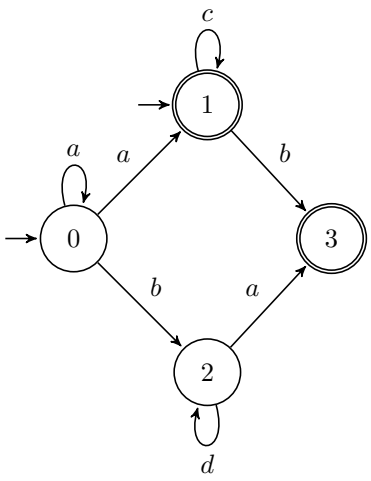
Théorème 11. *Les langages reconnaissables par automate fini sont représentables par expression rationnelle.*

On en présente ici une preuve constructive, connue sous le nom d'algorithme de McNaughton et Yamada.

Démonstration. Soit un automate A ayant comme ensemble d'états Q. L'idée va être de calculer, pour toute paire d'états (i, j), les ensembles de mots permettant d'aller de i à j en n'utilisant qu'une sélection d'états. On va faire ces calculs pour des sélections minimales, et grossir petit à petit, jusqu'à ne plus avoir de contrainte.

On introduit pour ça la notation $L^X_{i,j}$, qui représente sous forme d'expression rationnelle l'ensemble des mots non-vides menant de i à j en n'utilisant comme états intermédiaires que des états appartenant à l'ensemble X. Par exemple, dans l'automate

1. On entend souvent dire que la preuve d'un programme ou algorithme est au moins un ordre de grandeur plus complexe que l'écriture de ce dernier



$L_{0,3}^{\{1\}} = ac^*b$. En effet, on ne peut pas boucler en 0 car il servirait alors d'état intermédiaire, alors qu'il ne fait pas partie de ceux autorisés. De même, on ne peut pas passer par l'état 2. On peut en revanche boucler sur 1, d'où le c^* dans la regex.

On commence en calculer tous les $L_{i,j}^{\emptyset}$. Puisqu'on n'a pas le droit au moindre état intermédiaire, $L_{i,j}^{\emptyset}$ vaut la lettre étiquettant la transition de i à j si elle existe, et $\emptyset$ sinon. Dans notre exemple, on a donc

- En partant de 0 :
  - $L_{0,0}^{\emptyset} = a$
  - $L_{0,1}^{\emptyset} = a$
  - $L_{0,2}^{\emptyset} = b$
  - $L_{0,3}^{\emptyset} = \emptyset$
- En partant de 1 :
  - $L_{1,0}^{\emptyset} = \emptyset$
  - $L_{1,1}^{\emptyset} = c$
  - $L_{1,2}^{\emptyset} = \emptyset$
  - $L_{1,3}^{\emptyset} = b$
- En partant de 2 :
  - $L_{2,0}^{\emptyset} = \emptyset$
  - $L_{2,1}^{\emptyset} = \emptyset$
  - $L_{2,2}^{\emptyset} = d$
  - $L_{2,3}^{\emptyset} = a$
- En partant de 3 :
  - $L_{3,0}^{\emptyset} = \emptyset$
  - $L_{3,1}^{\emptyset} = \emptyset$
  - $L_{3,2}^{\emptyset} = \emptyset$
  - $L_{3,3}^{\emptyset} = \emptyset$

Maintenant qu'on a nos cas de base, il s'agit de passer aux étapes suivantes en ajoutant de nouveaux états. De façon générale, Si on a calculé tous les $L_{i,j}^{X}$ et qu'on veut s'autoriser à passer par l'état q, on va utiliser la formule

$$L_{i,j}^{X\cup\{q\}} = L_{i,j}^{X} \cup L_{i,q}^{X}(L_{q,q}^{X})^* L_{q,j}^{X}$$

L'idée est que, pour aller de i à j en ayant le droit d'utiliser les états de X et q, on passe ou non par q. Le deuxième cas est pris en compte la partie gauche de l'union ensembliste.

Supposons maintenant qu'on aille de i à j en passant au moins une fois par q. Dans ce cas, notre parcours commence par un premier chemin de i à q sans passer par q, cad. par la lecture d'un mot appartenant à $L_{i,q}^{X}$.

Le parcours peut passer plus d'une fois par q. On s'autorise donc à aller de q à q, ce qui correspond à la partie $(L_{q,q}^{X})^*$ de la formule.

Enfin, une fois qu'on a fait notre dernier passage par l'état q, il faut finir notre parcours en allant en j : $L_{q,j}^{X}$.

En résumé, pour calculer l'ensemble des mots menant de i à j en passant par les états de X ou q, on isole explicitement les éventuels passages par q pour pouvoir se ramener au cas précédent, où seuls les états appartenant à X sont autorisés. On peut donc maintenant avancer les calculs pour notre automate, par exemple en autorisant maintenant les passages par l'état 1 (l'ordre dans lequel on ajoute les états ne change pas le résultat, mais peut rendre les calculs plus ou moins longs) :

- En partant de 0 :
  - $L_{0,0}^{\emptyset\cup\{1\}} = L_{0,0}^{\emptyset} \cup L_{0,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,0}^{\emptyset} = a + a(c)^*\emptyset = a$ [2]
  - $L_{0,1}^{\{1\}} = L_{0,1}^{\emptyset} \cup L_{0,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,1}^{\emptyset} = a + ac^*c = a + ac^+ = ac^*$
  - $L_{0,2}^{\{1\}} = L_{0,2}^{\emptyset} \cup L_{0,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,2}^{\emptyset} = b + ac^*\emptyset = b$
  - $L_{0,3}^{\{1\}} = L_{0,3}^{\emptyset} \cup L_{0,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,3}^{\emptyset} = ac^*b$
- En partant de 1 :
  - $L_{1,0}^{\{1\}} = L_{1,0}^{\emptyset} \cup L_{1,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,0}^{\emptyset} = \emptyset$
  - $L_{1,1}^{\{1\}} = L_{1,1}^{\emptyset} \cup L_{1,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,1}^{\emptyset} = c + cc^*c = c^+$
  - $L_{1,2}^{\{1\}} = L_{1,2}^{\emptyset} \cup L_{1,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,2}^{\emptyset} = \emptyset$
  - $L_{1,3}^{\{1\}} = L_{1,3}^{\emptyset} \cup L_{1,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,3}^{\emptyset} = b + cc^*b = b + c^+b = c^*b$
- En partant de 2 :
  - $L_{2,0}^{\{1\}} = L_{2,0}^{\emptyset} \cup L_{2,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,0}^{\emptyset} = \emptyset$
  - $L_{2,1}^{\{1\}} = L_{2,1}^{\emptyset} \cup L_{2,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,1}^{\emptyset} = \emptyset$
  - $L_{2,2}^{\{1\}} = L_{2,2}^{\emptyset} \cup L_{2,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,2}^{\emptyset} = d$
  - $L_{2,3}^{\{1\}} = L_{2,3}^{\emptyset} \cup L_{2,1}^{\emptyset}(L_{1,1}^{\emptyset})^* L_{1,3}^{\emptyset} = a$
- En partant de 3, vu que $L_{3,j}^{\emptyset} = \emptyset$ pour tout état j, on n'arrivera jamais à aller plus loin. Donc $L_{3,j}^{\{1\}} = \emptyset$ pour tout j.

On continue en ajoutant 2 :

- En partant de 0 :
  - $L_{0,0}^{\{1\}\cup\{2\}} = L_{0,0}^{\{1\}} \cup L_{0,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,0}^{\{1\}} = a$
  - $L_{0,1}^{\{1,2\}} = L_{0,1}^{\{1\}} \cup L_{0,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,1}^{\{1\}} = ac^*$
  - $L_{0,2}^{\{1,2\}} = L_{0,2}^{\{1\}} \cup L_{0,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,2}^{\{1\}} = b + bd^*d = bd^*$
  - $L_{0,3}^{\{1,2\}} = L_{0,3}^{\{1\}} \cup L_{0,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,3}^{\{1\}} = ac^*b + bd^*a$

2. Pour rappel, $L\emptyset = \emptyset L = \emptyset$ pour tout langage L

— En partant de 1 :
  — $L_{1,0}^{\{1,2\}} = L_{1,0}^{\{1\}} \cup L_{1,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,0}^{\{1\}} = \emptyset$
  — $L_{1,1}^{\{1,2\}} = L_{1,1}^{\{1\}} \cup L_{1,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,1}^{\{1\}} = c^+$
  — $L_{1,2}^{\{1,2\}} = L_{1,2}^{\{1\}} \cup L_{1,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,2}^{\{1\}} = \emptyset$
  — $L_{1,3}^{\{1,2\}} = L_{1,3}^{\{1\}} \cup L_{1,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,3}^{\{1\}} = c^*b$
— En partant de 2 :
  — $L_{2,0}^{\{1,2\}} = L_{2,0}^{\{1\}} \cup L_{2,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,0}^{\{1\}} = \emptyset$
  — $L_{2,1}^{\{1,2\}} = L_{2,1}^{\{1\}} \cup L_{2,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,1}^{\{1\}} = \emptyset$
  — $L_{2,2}^{\{1,2\}} = L_{2,2}^{\{1\}} \cup L_{2,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,2}^{\{1\}} = d + dd^*d = d^+$
  — $L_{2,3}^{\{1,2\}} = L_{2,3}^{\{1\}} \cup L_{2,2}^{\{1\}}(L_{2,2}^{\{1\}})^* L_{2,3}^{\{1\}} = a + dd^*a = d^*a$
— $L_{3,j}^{\{1,2\}} = \emptyset$ pour tout j.

On ajoute maintenant 0 :

— En partant de 0 :
  — $L_{0,0}^{\{1,2\}\cup\{0\}} = L_{0,0}^{\{1,2\}} \cup L_{0,0}^{\{1,2\}}(L_{0,0}^{\{1,2\}})^* L_{0,0}^{\{1,2\}} = a + aa^*a = a^+$
  — $L_{0,1}^{\{0,1,2\}} = L_{0,1}^{\{1,2\}} \cup L_{0,0}^{\{1,2\}}(L_{0,0}^{\{1,2\}})^* L_{0,1}^{\{1,2\}} = ac^* + aa^*ac^* = a^+c^*$
  — $L_{0,2}^{\{0,1,2\}} = L_{0,2}^{\{1,2\}} \cup L_{0,0}^{\{1,2\}}(L_{0,0}^{\{1,2\}})^* L_{0,2}^{\{1,2\}} = bd^* + aa^*bd^* = a^*bd^*$
  — $L_{0,3}^{\{0,1,2\}} = L_{0,3}^{\{1,2\}} \cup L_{0,0}^{\{1,2\}}(L_{0,0}^{\{1,2\}})^* L_{0,3}^{\{1,2\}} = ac^*b + bd^*a + aa^*(ac^*b + bd^*a) = a^*(ac^*b + bd^*a)$
— Comme on en a un peu marre, on remarque que $L_{i,0}^{\{1,2\}} = \emptyset$ quand $i \neq 0$, et donc que $L_{i,0}^{\{1,2\}}(L_{0,0}^{\{1,2\}})^* L_{0,j}^{\{1,2\}} = \emptyset$ pour tout état j. Donc, pour les départs d'états autres que 0, le fait de pouvoir passer par 0 ne va rien pouvoir ajouter. On a donc $L_{i,0}^{\{0,1,2\}} = L_{i,0}^{\{1,2\}}$ pour $i \neq 0$.

De même, $L_{3,i}^{\{0,1,2\}} = \emptyset$ pour tout état i. On a donc $L_{i,j}^{Q} = L_{i,j}^{\{0,1,2\}}$ pour toute paire d'états i et j.

Maintenant qu'on a calculé les langages reconnus par toute paire d'états, on combine ceux qui nous intéressent. Puisque les états initiaux sont 0 et 1 et que les terminaux sont 1 et 3, le langage reconnu par l'automate est l'union des mots allant de 0 à 1, de 0 à 3, de 1 à 1 et de 1 à 3 (on fait toutes les combinaisons d'état initial / état terminal).

Attention cependant, $L_{i,j}^{Q}$ calcule l'ensemble des mots <u>non-vides</u> allant de i à j. Donc si un état est à la fois initial et terminal, il faut ajouter à la main le mot vide.

En conclusion, le langage reconnu par l'automate de l'exemple est décrit par la regex

$$L_{0,1}^{Q} + L_{0,3}^{Q} + L_{1,1}^{Q} + L_{1,3}^{Q} + \epsilon = a^+c^* + a^*(ac^*b + bd^*a) + c^+ + c^*b + \epsilon$$

□

Etant donné un automate fini, l'algorithme de McNaughton et Yamada permet donc, bien que parfois quelque peu laborieusement, de produire une expression rationnelle décrivant le même langage, réalisant donc la deuxième partie du théorème de Kleene.

Chapitre 5

Grammaires formelles et hiérarchie de Chomsky

5.1 Principe général

Une **grammaire formelle** est une série de **règles** permettant de générer des mots. Ces règles utilisent des **symboles** dits **terminaux** (les lettres "normales", par convention en minuscules), et d'autres dits **non-terminaux** (normalement dénotés par des lettres majuscules). Un de ces symboles non-terminaux, appelé **axiome**, indique le début de toute génération de mot. On présente d'abord le fonctionnement des grammaires et les concepts de base à l'aide de quelques exemples.

Exemple 5.1.1. *On présente la grammaire dénotée par ces deux règles :*

$$\begin{cases} S \to \epsilon \\ S \to abS \end{cases}$$

Cette grammaire contient un seul symbole non-terminal, S. Il s'agit donc automatiquement de l'axiome. Le symbole S peut se transformer en ϵ (première règle) ou en abS (deuxième règle). Une grammaire génère l'ensemble des mots composés uniquement de symboles terminaux (ici a et b) qu'on peut obtenir en partant de l'axiome et en appliquant autant de fois qu'on veut les règles données. Dans ce qui suit, on écrira $\to_1$ pour une application de la première règle et $\to_2$ pour la seconde.

On peut par exemple obtenir le mot ab de la façon suivante :

$$S \to_2 abS \to_1 ab$$

Dans cette suite de transformation, appelée ***dérivation****, on remplace d'abord l'axiome par abS à l'aide de la deuxième règle. Puisque abS contient le facteur S, on peut utiliser localement la deuxième règle pour faire disparaître ce S. Dans ce cas, ce qu'il y avait autour du S, le* ***contexte****, reste inchangé.*

On peut également obtenir le mot abab :

$$S \to_2 abS \to_2 ababS \to_1 abab$$

ou abababab :

$$S \to_2 abS \to_2 ababS \to_2 abababS \to_1 abababab$$

Il n'est pas obligatoire d'utiliser toutes les règles d'une grammaire. On peut donc générer le mot vide :

$$S \to_1 \epsilon$$

On se rend compte assez vite que la grammaire ***engendre*** *le langage* $(ab)^*$.

Exemple 5.1.2. *Un symbole non-terminal peut générer d'autres symboles non-terminaux, et même plusieurs en même temps. De plus, un non-terminal peut générer un autre mot que juste* ϵ*. Ces deux points sont illustrés par la grammaire suivante :*

$$\begin{cases} S \to LaaR \\ L \to Lb \\ L \to ab \\ R \to bR \\ R \to ba \end{cases}$$

Quelques exemples de dérivations dans cette grammaire :

$$S \to_1 LaaR \to_2 LbaaR \to_2 LbbaaR \to_4 LbbaabR \to_3 abbbaabR \to_5 abbbaabba$$

$$S \to_1 LaaR \to_3 abaaR \to_5 abaaba$$

On note $\to_i^j$ *j utilisations de la règle numéro i, comme dans la dérivation suivante :*

$$S \to_1 LaaR \to_2^5 LbbbbbaaR \to_3 abbbbbbaaR \to_4^3 abbbbbbaabbbR \to_5 abbbbbbaabbbba$$

Remarque Quand on a plusieurs règles de la forme

$$\begin{cases} u \to v_1 \\ \dots \\ u \to v_n \end{cases}$$

on pourra gagner de la place en écrivant

$$\{u \to v_1 \mid \dots \mid v_n$$

Par exemple, la grammaire de l'exemple 5.1.2 peut également s'écrire

$$\begin{cases} S \to LaaR \\ L \to Lb \mid ab \\ R \to bR \mid ba \end{cases}$$

Dans ce cas, $L \to Lb$ est la deuxième règle de la grammaire, tandis que $L \to ab$ en est la troisième.

Exercice 5.1.1. *Quel est le langage engendré par la grammaire de l'exemple 5.1.2 ?*

Exercice 5.1.2. *Quel est le langage engendré par la grammaire suivante ?*

$$\begin{cases} S \to A \mid B \\ A \to aA \mid \epsilon \\ B \to bB \mid \epsilon \end{cases}$$

Exercice 5.1.3. *Quel est le langage engendré par la grammaire suivante ?*

$$\begin{cases} S \to aS \mid bS \mid \epsilon \end{cases}$$

Exercice 5.1.4. *Donner l'ensemble des mots qui admettent deux dérivations (ie. peuvent être construits de plusieurs façons différentes) dans la grammaire de l'exercice 5.1.2. Même question pour celle de l'exercice 5.1.3.*

Exemple 5.1.3. *On a jusqu'ici seulement vu des exemples de grammaires avec un seul non-terminal à gauche des flèches de réécriture. Il est cependant possible de préciser le contexte dans lequel les réécritures doivent se faire, comme dans la grammaire suivante :*

$$\begin{cases} S \to SABC \mid \epsilon \\ AB \to BA \\ BA \to AB \\ AC \to CA \\ CA \to AC \\ BC \to CB \\ CB \to BC \\ A \to a \\ B \to b \\ C \to c \end{cases}$$

Toute dérivation dans cette grammaire fonctionne de la façon suivante : on commence par utiliser la première règle n fois pour produire $S(ABC)^n$. Ensuite, on utilise la deuxième règle pour faire disparaître S et se retrouver avec $(ABC)^n$. Les règles 3 à 8 permettent ensuite de mélanger les symboles non-terminaux comme on l'entend. Une fois qu'ils ont été disposés de la façon voulue, on les transforme en a, b et c avec les règles 9, 10 et 11.

Le langage engendré par cette grammaire est donc celui des mots contenant autant de a que de b et de c.

Remarque Dans l'exemple ci-dessus, on découpe toute dérivation en 4 étapes : utilisation de la première règle, puis de la seconde, puis des 3 à 8, et enfin des 9 à 11. Cette présentation nous semble améliorer la compréhension de l'exemple et du langage engendré, mais techniquement, rien n'empêche de mélanger les étapes, comme dans la dérivation suivante :

$$S \to_1 SABC \to_9 SaBC \to_7 SaCB \to_{11} SaCb \to_1 SABCaCb \to_2 ABCaCB \to^5_{9,10,11} abcacb$$

Où $\to^5_{9,10,11}$ indique 5 utilisations de règles parmi 9, 10 et 11.

5.2 Formalisation

Maintenant qu'on a un peu joué avec les grammaires formelles, on peut regarder comment elles se formalisent. Techniquement donc, une grammaire formelle est un quadruplet $\langle \Sigma, V, S, R \rangle$, où

- Σ est l'ensemble des **symboles terminaux**
- V est l'ensemble des **symboles non-terminaux**
- $S \in V$ est l'**axiome**
- R est l'ensemble des **règles de production**, ou règles de réécriture. Ces dernières forment un sous-ensemble de $(\Sigma \cup V)^* V (\Sigma \cup V)^* \times (\Sigma \cup V)^*$, cad. qu'elles doivent toujours avoir au moins un non-terminal à gauche.

Exemple 5.2.1. *La grammaire de l'exemple 5.1.2 s'écrit*

$$\langle \{a, b\}, \{S, L, R\}, S, \begin{cases} S \to LaaR \\ L \to Lb \mid ab \\ R \to bR \mid ba \end{cases} \rangle$$

Définition 5.2.1: Réécriture / dérivation immédiate

Soient p, s et g des mots sur l'alphabet $(\Sigma \cup V)$ (ils sont donc potentiellement vides), et $f \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$. Si la règle r est de la forme $f \to g$, alors le mot pfs se ***réécrit*** *(ou dérive)* ***immédiatement*** *en pgs via la règle r, ce qu'on note*

$$pfs \to_r pgs$$

Dit autrement, si le mot f apparaît dans le contexte p_s, alors on peut transformer f en g en gardant le contexte.
Plus généralement, étant donnée une grammaire G, on dit qu'elle réécrit (ou dérive) immédiatement un mot u en v ssi. il existe une règle r dans la grammaire telle que u se réécrit immédiatement en v via la règle r. On écrit ceci

$$pfs \to_G pgs$$

Quand il n'y a pas d'ambiguïté sur la grammaire étudiée, on pourra se permettre de ne pas noter le G et écrire

$$pfs \to pgs$$

Définition 5.2.2: Réécriture / dérivation

Soient une grammaire G et deux mots u et v. On dit que u se **réécrit** (ou dérive) en v avec G ssi. il existe une série (potentiellement nulle) de réécritures immédiates menant de u à v dans G. On le note

$$u \to_G^* v$$

ou $u \to^* v$ s'il n'y a pas d'ambiguïté sur la grammaire étudiée.
Pour une série d'au moins une dérivation, on écrira $u \to_G^+ v$ ou $u \to^+ v$

Exemple 5.2.2. *Dans l'exemple 5.1.3, on a*

$$S \to^* SABCaCb \to^* abcacb$$

et donc

$$S \to^* abcacb$$

Définition 5.2.3: Langage engendré

Le langage engendré par le mot f dans une grammaire G, noté $L_G(f)$, est l'ensemble des mots de Σ^* (formés uniquement de symboles terminaux) en lesquelles on peut réécrire l'axiome de f en suivant les règles de G. Plus formellement,

$$L_G(f) = \{u \in \Sigma^* \mid f \to_G^* u\}$$

Le langage engendré par une grammaire est le langage engendré par l'axiome dans cette grammaire. Soit une grammaire G d'axiome S, son langage engendré, noté L_G, est $L_G(S)$.

Exemple 5.2.3. *Soit G la grammaire de l'exemple 5.1.3.*

$$L_G = \{w \in \Sigma^* \mid |w|_a = |w|_b = |w|_c\}$$

Exercice 5.2.1. *Donner une grammaire qui engendre le langage $\{a^n b^n \mid n \in \mathbb{N}\}$*

Exercice 5.2.2. *Montrer que la grammaire suivante engendre le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$*

$$\begin{cases} S \to XY \\ X \to aXbZ \mid \epsilon \\ Zb \to bZ \\ ZY \to Yc \\ Y \to \epsilon \end{cases}$$

5.3 Hierarchie de Chomsky

Chomsky propose en 1956 une classification des grammaires en fonction des formes de leurs règles. Les grammaires sont donc divisées en 4 catégories, appelées grammaires de type 0, 1, 2 et 3.

5.3.1 Grammaires de type 3, ou régulières

Les grammaires de type 3, appelées **grammaires régulières**, se divisent elles-même en deux sous-classes : les **grammaires linéaires à gauche** et **grammaires linéaires à droite**.

Grammaires linéaires à gauche et à droite

Dans les deux cas, on ne permet que d'écrire un symbole terminal à la fois, en allant toujours dans le même sens.

Définition 5.3.1: Grammaire linéaire à gauche

Une grammaire linéaire à gauche est une grammaire dont toutes les règles sont de la forme
$A \to Ba$
ou
$A \to a$
ou
$A \to \epsilon$
où A et B sont des symboles non-terminaux, et a un terminal.

Exemple 5.3.1. *La grammaire*

$$\begin{cases} S \to Sa \mid Ta \mid a \\ T \to Tb \mid b \end{cases}$$

est une grammaire linéaire à gauche.

Exercice 5.3.1. *Quel est le langage engendré par la grammaire de l'exemple 5.3.1 ?*

Définition 5.3.2: Grammaire linéaire à droite

Une grammaire linéaire à droite est une grammaire dont toutes les règles sont de la forme
$A \to aB$
ou
$A \to a$
ou
$A \to \epsilon$
où A et B sont des symboles non-terminaux, et a un terminal.

Exemple 5.3.2. *La grammaire*

$$\begin{cases} S \to aS \mid bS \mid \epsilon \end{cases}$$

est une grammaire linéaire à droite.

Remarque Les grammaires de type 3, ou régulières, est l'ensemble des grammaires linéaires à gauche et des grammaires linéaires à droite, mais sans mélange. Par exemple, la grammaire

$$\begin{cases} S \to aS \mid Sb \mid \epsilon \end{cases}$$

n'est **pas** une grammaire de type 3.

Lemme 7. *Les grammaires linéaires à gauche et les grammaires linéaire à droite ont la même expressivité. Dit autrement, tout langage engendré par une grammaire linéaire à gauche est est également engendré par une grammaire linéaire à droite, et inversement.*

Démonstration. On n'entrera pas ici dans les détails, qui dépassent le cadre de ce cours. Etant donnée une grammaire linéaire à gauche (resp. droite), on peut écrire une grammaire linéaire à

droite (resp. gauche) qui la simule en partant des règles $A \to a$ et $A \to \epsilon$. Les règles *normales* sont inversées et nouvelles règles *terminales* correspondent à l'axiome de la première grammaire. □

Exercice 5.3.2. *(**) Donner, en suivant l'intuition de la preuve ci-dessus, une grammaire linéaire à droite qui engendre le même langage que la grammaire de l'exemple 5.3.1.*

Une extension du théorème de Kleene

On a vu dans le chapitre 4 que les langages pouvant être décrits par des expressions rationnelles et ceux définissables par automate étaient les mêmes. Ce résultat peut s'étendre aux grammaires de type 3.

Théorème 12. *Les langages reconnus par automate fini sont exactement ceux engendrés par des grammaires de type 3.*

Démonstration. Comme dans le théorème de Kleene original, on propose comme preuve une traduction des grammaires vers les automates, et inversement. Contrairement aux expressions rationnelles, les grammaires de type 3 ont un fonctionnement très similaire aux automates finis, les traductions sont donc très simples.

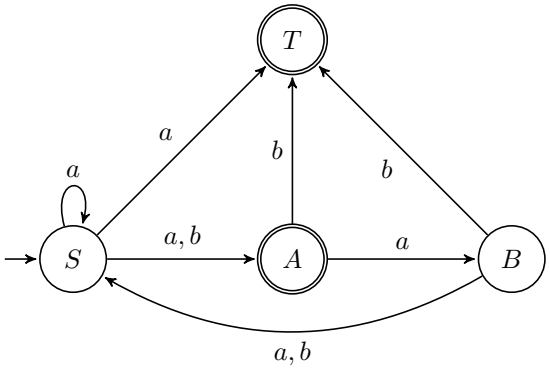
Soit G une grammaire de type 3. Puisque les grammaires linéaires à gauche et à droite engendrent les mêmes langages, on peut supposer que G est une grammaire linéaire à droite. On crée un automate dont les états sont les symboles non-terminaux de G, ainsi qu'un état supplémentaire et terminal T (pour terminus). L'état initial est son axiome. Toute règle $A \to aB$ devient une transition de A vers B en lisant a. Les règles $A \to a$ deviennent des transitions de A vers T en a. Enfin, on rend terminal tout état A tel qu'il existe une règle $A \to \epsilon$.

Soit A un automate fini. On déterminise A en A_{det}. On crée une grammaire dont les symboles non-terminaux sont les états de A_{det}. L'axiome est l'état initial de A_{det}. Toute transition $A \xrightarrow{a} B$ se traduit par une règle $A \to aB$. Pour tout état terminal A, on ajoute une règle $A \to \epsilon$. □

Exemple 5.3.3. *Soit la grammaire suivante :*

$$\begin{cases} S \to aS \mid aA \mid bA \mid a \\ A \to aB \mid b \mid \epsilon \\ B \to b \mid aS \mid bS \end{cases}$$

Le langage qu'elle engendre est reconnu par l'automate suivant :



Exemple 5.3.4. *L'automate de l'exemple 3.5.4 est simulé par la grammaire qui suit, où 0 est remplacé par A, 1 par B etc, et A est l'axiome.*

$$\begin{cases} A \to bA \mid aB \mid \epsilon \\ B \to aB \mid bC \mid \epsilon \\ C \to aD \mid bA \mid \epsilon \\ D \to aD \mid bD \end{cases}$$

Exercice 5.3.3. *Traduire en automate la grammaire*

$$\begin{cases} S \to aS \mid bB \mid a \mid \epsilon \\ A \to aS \mid \epsilon \\ B \to a \end{cases}$$

Exercice 5.3.4. *Traduire en grammaire l'automate de l'exemple 3.3.2.*

5.3.2 Grammaires de type 2, ou non contextuelles

Les grammaires de type 2, appelées **grammaires non contextuelles** (*context-free*), ou parfois grammaires algébriques, n'ont aucune contrainte quant à la partie droite des règles. Elles imposent cependant que la partie gauche ne contienne pas plus d'un caractère, empêchant donc de prendre en compte le contexte dans les dérivations.

Définition 5.3.3: Grammaire non contextuelle

Une grammaire non contextuelle est une grammaire dont toutes les règles sont de la forme
$$A \to \gamma$$
où A est un symbole non-terminal, et γ est n'importe quel mot constitué de symboles terminaux et non-terminaux.

Exemple 5.3.5. *La grammaire*

$$\begin{cases} S \to aSb \\ S \to \epsilon \end{cases}$$

est une grammaire de type 2, qui engendre le langage $\{a^n b^n \mid n \in \mathbb{N}\}$

Remarque Toute grammaire de type 3 est également une grammaire de type 2, puisque toute règle d'une grammaire de type 3 a bien un seul symbole à gauche de toute règle.

Lemme 8. *Les langages engendrés par des grammaires de type 3 forment un sous-ensemble strict des langages engendrés par des grammaires de type 2. Autrement dit, si un langage est engendré par une grammaire de type 3, alors il l'est également par une grammaire de type 2, mais il existe au moins un (et même plein) langage engendré par une grammaire de type 2 qui ne peut pas être engendré par une grammaire de type 3.*

Démonstration. L'inclusion non-stricte est triviale avec la remarque précédente. En effet, soit L un langage engendré par la grammaire de type 3 G, alors G est également une grammaire de type 2 qui, de fait, engendre toujours L.

Pour l'aspect strict de cette inclusion, la grammaire de type 2 de l'exemple 5.3.5 engendre le langage $\{a^n b^n \mid n \in \mathbb{N}\}$. Or, on a vu en 3.4 que ce langage n'était pas reconnaissable par automate. Puisque les grammaires de type 3 ont la même expressivité que ces derniers (théorème 12), il n'en existe pas qui engendre ce langage. □

5.3.3 Grammaires de type 1, ou contextuelles

Les grammaires de type 1, appelées **grammaires contextuelles** (*context-sensitive*) permettent d'encadrer le non-terminal à gauche de toute règle par un *contexte*, qui doit apparaître tel quel à droite. De plus, le non-terminal ne peut pas être réécrit en ϵ, sauf cas particulier pour capturer le mot vide.

Définition 5.3.4: Grammaire contextuelle

Une grammaire contextuelle est une grammaire dont toutes les règles sont de la forme
$\alpha X \beta \to \alpha \gamma \beta$, où α, β et γ sont des mots constitués de symboles terminaux et non-terminaux, X est un symbole non-terminal et γ est non-vide.
ou
$S \to \epsilon$, où S est l'axiome de la grammaire. Si cette règle est utilisée, S n'a pas le droit d'apparaître à droite d'une règle.

Cette exception un peu étrange permet aux grammaires contextuelles d'engendrer le mot vide, ce qui ne serait pas possible uniquement avec le premier type de règle. Supposons par exemple qu'on ait une grammaire contextuelle G d'axiome S engendrant un langage L ne contenant pas le mot vide, on peut facilement construire la grammaire G' qui engendre $L \cup \{\epsilon\}$.

Il suffit d'ajouter un nouveau non-terminal, S', qui sera le nouvel axiome, ainsi que les deux règles qui suivent :

$$\begin{cases} S' \to \epsilon \\ S' \to S \end{cases}$$

Ainsi, on permet d'engendrer ϵ dès le début, ou de faire une génération "normale" de G en repartant de S. Avec cette transformation, on est sûr que S' n'apparaît jamais à droite d'une règle.

Exemple 5.3.6. *La grammaire de l'exercice 5.2.2* ***n'est pas*** *une grammaire de type 1, à cause des règles $X \to \epsilon$ et $Y \to \epsilon$. On peut cependant légèrement la modifier pour obtenir une grammaire de type 1 qui engendre le même langage, c'est-à-dire $\{a^n b^n c^n \mid n \in \mathbb{N}\}$*

$$\begin{cases} S' \to S \mid \epsilon \\ S \to aSBC \mid aBC \\ CB \to HB \\ HB \to HC \\ HC \to BC \\ aB \to ab \\ bB \to bb \\ bC \to bc \\ cC \to cc \end{cases}$$

Notez en particulier l'utilisation des règles

$$\begin{cases} CB \to HB \\ HB \to HC \\ HC \to BC \end{cases}$$

Pour "simuler"

$$\begin{cases} CB \to BC \end{cases}$$

qui n'est pas autorisée dans une grammaire contextuelle. En effet, soit on considère que C est le contexte et B le non-terminal réécrit, auquel cas le C à droite est du mauvais côté, soit on inverse les rôles, et dans ce cas le B obtenu est également du mauvais côté.

Exercice 5.3.5. *Montrer que la grammaire de l'exemple 5.3.6 engendre bien le langage* $\{a^n b^n c^n \mid n \in \mathbb{N}\}$.

Remarque Une grammaire de type 2 **n**'est **pas forcément** une grammaire de type 1, à cause des règles de la forme $A \to \epsilon$. Il existe cependant un algorithme qui prend une grammaire de type 2 et en renvoie une autre, également de type 2, engendrant le même langage sans ϵ-production, sauf éventuellement pour engendrer le mot vide. Cet algorithme sera décrit en 5.4.2.

Lemme 9. *Les langages engendrés par des grammaires de type 2 forment un sous-ensemble strict des langages engendrés par des grammaires de type 1. Autrement dit, si un langage est engendré par une grammaire de type 2, alors il l'est également par une grammaire de type 1, mais il existe au moins un (et même plein) langage engendré par une grammaire de type 1 qui ne peut pas être engendré par une grammaire de type 2.*

Démonstration. L'inclusion non-stricte est encore une fois simple avec la remarque précédente. En effet, soit L un langage engendré par la grammaire G de type 2, alors on peut construire G' une grammaire de type 2 qui est également de type 1 et engendre toujours L.

Pour l'aspect strict de cette inclusion, la grammaire de type 1 de l'exemple 5.3.6 engendre le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. On n'en donnera malheureusement pas la preuve ici, mais ce langage n'est pas engendrable par une grammaire de type 2 [1]. □

1. Vous pouvez cependant essayer de vous en convaincre en essayant d'écrire une telle grammaire !

Remarque Les grammaires contextuelles sont également capables d'engendrer $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$, et ainsi de suite, à l'aide de grammaires de plus en plus complexes.

5.3.4 Grammaires de type 0, ou générales

Les grammaires de type 0, appelées **grammaires générales**, sont tout simplement l'ensemble des grammaires formelles. Elles n'ont donc aucune contrainte, sinon toujours celle d'avoir au moins un symbole non-terminal à gauche de toute règle. Il va sans dire que toute grammaire présenté jusqu'ici ou dans la suite de ce document constitue une grammaire de type 0.

Lemme 10. *Les langages engendrés par des grammaires de type 1 forment un sous-ensemble strict des langages engendrés par des grammaires de type 0. Autrement dit, si un langage est engendré par une grammaire de type 1, alors il l'est également par une grammaire de type 0, mais il existe au moins un (et même plein) langage engendré par une grammaire de type 0 qui ne peut pas être engendré par une grammaire de type 1.*

Démonstration. L'inclusion non-stricte est triviale, puisque toute grammaire de type 1 est également une grammaire de type 0.

Pour l'aspect strict de cette inclusion, c'est un peu plus compliqué, puisqu'il n'existe pas - à notre connaissance - d'exemple très concret et naturel de langage strictement engendrable par une grammaire de type 0. On renverra par contre à cet exemple abstrait et très amusant [2]. Informellement, l'idée est de faire une liste (infinie) des mots sur un alphabet, ainsi qu'une liste (également infinie) des grammaires de type 1 sur le même alphabet (c'est un peu technique, mais on peut écrire un algorithme qui parcourt cet ensemble sans rien oublier).

On aligne ces deux listes, cad. qu'on a maintenant une liste de paires (un mot et une grammaires, associés arbitrairement). On définit maintenant L le langage des mots qui n'appartiennent pas au langage engendré par la grammaire qui leur est associée. Sans entrer dans les détails, ce langage est bien engendré par une grammaire de type 0 consistant en gros en un "simulateur de grammaire de type 1".

Il n'est par contre pas engendré par une grammaire de type 1. En effet, si une telle grammaire G existait, elle serait à une position j de la liste des grammaires. Dans ce cas, le j^{eme} mot appartient à L ssi. il n'appartient pas à L_G. La grammaire G ne peut donc en fait pas engendrer L. □

5.4 Utilisation de grammaires algébriques

On finit ce chapitre en présentant quelques spécificités des grammaires algébriques, ou de type 2.

5.4.1 Arbres de dérivation

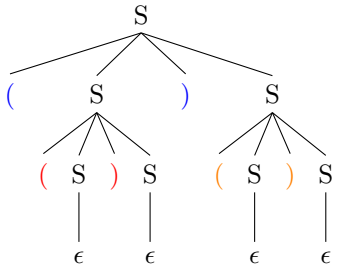
Puisque les grammaires algébriques ne peuvent pas avoir plus d'un symbole à gauche de leurs règles, leurs dérivations se représentent efficacement sous forme d'arbre, où chaque noeud interne est un symbole non-terminal, dont les descendants sont les symboles produits par application d'une règle. Le mot engendré est alors la concaténation, de la gauche vers la droite (parcours préfixe) des symboles aux feuilles.

2. https ://cs.stackexchange.com/a/56634

Exemple 5.4.1. *Soit la grammaire suivante, où "(" et ")" sont des symboles terminaux :*

$$\begin{cases} S \to (S)S \mid \epsilon \end{cases}$$

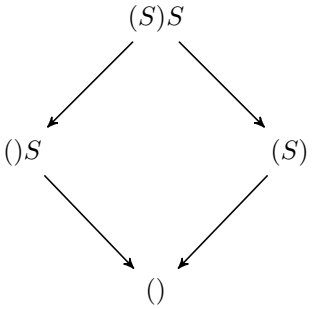
Le mot (())() peut par exemple être dérivé de la façon suivante :



Exercice 5.4.1. *Quel est le langage engendré par la grammaire de l'exemple 5.4.1 ?*

Exercice 5.4.2. *Donner un arbre de dérivation du mot abbbaaba dans la grammaire de l'exemple 5.2.1.*

Cette représentation n'indique pas l'ordre dans lequel les règles sont appliquées. Mais vu que les règles des grammaires algébriques ne peuvent pas prendre en compte le contexte, cet ordre importe peu, puisqu'on reviendra toujours au même, par exemple



En plus d'être très lisible, les arbres de dérivations donnent une décompostion structurelle du mot dérivé. La structure d'un mot (ou d'une phrase) est évidemment cruciale pour l'interprétation. Soit par exemple la grammaire suivante, où "+", "×", "(" et ")" sont des symboles terminaux et n représente n'importe quel nombre.

$$\begin{cases} S \to S + S \mid S \times S \mid (S) \mid n \end{cases}$$

Sans définir de priorité sur les opérateurs, le mot $n + n \times n$ est ambigü. Mais, si on en regarde la dérivation (cf. la figure 5.1), on comprend s'il s'agit de $n + (n \times n)$ (arbre gauche) ou $(n + n) \times n$ (arbre droit).

FIGURE 5.1 – Deux dérivations de $n + n \times n$

Pour des exemples plus linguistiques, on renvoie évidemment le lecteur ou la lectrice à ses cours de syntaxe.

Définition 5.4.1: Grammaire algébrique ambiguë

Une grammaire algébrique est dite **ambiguë** ssi. son langage engendré contient au moins un mot admettant deux arbres de dérivation différents.

Définition 5.4.2: Pouvoir génératif

Deux grammaires algébriques ont le même **pouvoir génératif faible** ssi elles engendrent les mêmes mots, mais pas forcément avec la même structure. Elles ont le même **pouvoir génératif fort** ssi. elles engendrent des arbres équivalents.

Exemple 5.4.2. *La grammaire*

$$\begin{cases} S \to S + T \mid S \times T \mid T \\ T \to (S) \mid n \end{cases}$$

n'est pas ambiguë : le mot $n + n \times n$ a une seule dérivation, où l'addition est effectuée avant la multiplication[3]*. Si on veut que la multiplication soit effectuée avant l'addition, on est obligé de faire apparaître des parenthèses :*

3. Cette grammaire n'est donc pas une bonne représentation de l'arithmétique standard, puisqu'elle n'en respecte pas les priorités. Une bonne grammaire algébrique faisant ça existe, mais est plus compliquée. En pratique, si vous écrivez avec un outil comme yacc un parser sous la forme d'une grammaire algébrique, vous pourrez écrire une telle grammaire de façon naïve en précisant les priorités entre opérateurs, et une bonne grammaire sera générée automatiquement.

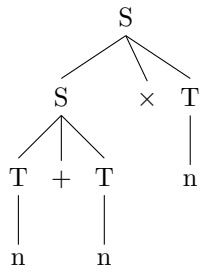


FIGURE 5.2 – Dérivation de $n + n \times n$

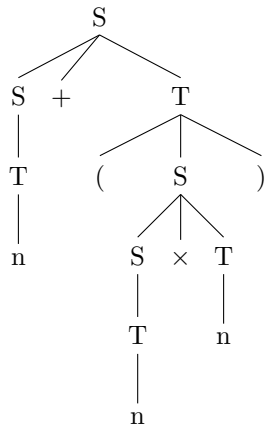


FIGURE 5.3 – Dérivation de $n + (n \times n)$

Les deux grammaires d'opérations arithmétiques présentées ont le même pouvoir génératif faible, mais pas fort.

Définition 5.4.3: Langage intrinsèquement ambigu

Un langage algébrique (ie. engendré par une grammaire algébrique) est **intrinsèquement ambigu** ssi. toutes les grammaires algébriques le reconnaissant sont ambiguës.

Exemple 5.4.3. *Le langage des opérations arithmétiques décrit plus haut est engendré par au moins une grammaire ambiguë, mais aussi une non-ambiguë. Ce langage n'est donc pas intrinsèquement ambigu.*

Exemple 5.4.4. *Le langage*

$$\{a^n b a^m b a^p b a^q \mid n, m, p, q \in \mathbb{N} \text{ et } ((n \geq m \text{ et } p \geq q) \text{ ou } (n \geq q \text{ et } m \geq p))\}$$

est intrinsèquement ambigu. Voir le DM2 pour la justification.

5.4.2 Transformation de grammaires algébriques

On présente ici quelques algorithmes pour nettoyer des grammaires algébriques.

Définition 5.4.4: Grammaire algébrique propre

Une grammaire algébrique est dite **propre** ssi. elle est **ϵ-libre**, **sans cycle** et **dépourvue de symboles inutiles**.

Définition 5.4.5: Grammaire ϵ-libre

Une grammaire est **ϵ-libre** ssi. elle ne possède pas d'ϵ-production, cad. de règle de la forme $\alpha \to \epsilon$, sauf éventuellement $S \to \epsilon$, auquel cas l'axiome S ne doit jamais apparaître à droite d'une règle.

Définition 5.4.6: Grammaire sans cycle

Une grammaire est **sans cycle** ssi. elle ne permet pas de dérivation de la forme $A \to^+ A$, où A est un symbole non-terminal.

Exemple 5.4.5. *La grammaire*

$$\begin{cases} S \to AB \mid C \\ A \to C \\ B \to \epsilon \\ C \to S \mid a \end{cases}$$

possède des cycles en S et A :

$$S \to AB \to A \to C \to S$$

$$A \to C \to S \to AB \to A$$

Définition 5.4.7: Symbole inutile

Un symbole inutile est un symbole non-terminal qui est soit **inaccessible**, soit **sans-production**.
Un symbole A est inaccessible ssi. il n'existe pas de dérivation $S \to^* \alpha A \beta$, cad. qu'on ne peut pas produire de A en partant de l'axiome.
Un symbole A est sans production ssi. il n'existe pas de dérivation $A \to^* w$ où $w \in \Sigma^*$, cad. qu'on ne peut pas dériver un "vrai mot" à partir de A.

Exemple 5.4.6. *La grammaire*

$$\begin{cases} S \to A \mid B \\ A \to CE \\ B \to \epsilon \\ C \to a \\ D \to Fa \mid \epsilon \\ E \to abbA \\ F \to bD \end{cases}$$

possède deux symboles inutiles. En effet, A est sans production (après en avoir produit un, on doit "résoudre" un E, qui lui-même produit un A et ainsi de suite), et D et F sont inaccessibles.

Elimination d'ϵ-productions

On doit d'abord calculer l'ensemble K des symboles non-terminaux A tels que $A \to^* \epsilon$, cad. les symboles qui peuvent se réécrire en ϵ.

Déjà, pour toute règle de la forme $A \to \epsilon$, on met évidemment A dans K. On procède ensuite par point fixe, en inspectant les différentes règles. A chaque règle $B \to \alpha$, si α est composé uniquement de symboles de K, alors α peut se réécrire en ϵ, on ajoute donc B à K. On refait une passe sur l'ensemble des règles tant qu'on a des ajouts à K.

Exemple 5.4.7. *Soit la grammaire*

$$\begin{cases} S \to CD \mid AB \mid E \\ A \to a \mid BC \\ B \to abba \mid \epsilon \\ C \to BB \\ D \to Sa \\ E \to AaBEB \mid BSb \end{cases}$$

Le symbole B peut se réécrire directement en ϵ, on l'ajoute donc à K. On fait maintenant une première passe sur les différentes règles et on remarque que C peut se réécrire en BB. Puisque $B \in K$, B peut se réécrire en ϵ, ce qui implique que BB peut se réécrire en $\epsilon\epsilon = \epsilon$. On ajoute donc C à K.

Puisqu'on a eu un ajout à K, on refait une passe sur les règles. A peut se réécrire en BC, donc en ϵ. On ajoute A à K.

Nouvelle passe, on ajoute S à K grâce à la règle $S \to AB$.

Enfin, notre dernière passe ne révèle pas de nouveau symbole à ajouter à K. On a donc atteint notre point fixe $K = \{S, A, B, C\}$.

Puisqu'on veut effacer les règles de la forme $A \to \epsilon$ tout en engendrant le même langage, on doit modifier les règles pour rattraper les mots éventuellement perdus. L'astuce va être de dupliquer toute règle en "remplaçant potentiellement" chaque symbole de K qu'elle contient par ϵ.

Exemple 5.4.8. *On reprend la grammaire et l'ensemble K de l'exemple 5.4.7.*

On commence par la règle $S \to CD$. Puisque C appartient à K, il peut se réécrire en ϵ, on intègre donc cette possibilité en ajoutant une règle $S \to D$. De même, la règle $S \to AB$ va être dupliquée en $S \to A$ et $S \to B$. Les symboles A et B peuvent se réécrire en ϵ lors d'une même dérivation, mais puisque le but de l'algorithme est d'éliminer les ϵ-productions, on ne va évidemment pas ajouter de règle $S \to \epsilon$.

Pour les mêmes raisons, on va ajouter les règles $A \to B$, $A \to C$, $C \to B$, $D \to a$.

La règle $E \to AaBEC$ contenant 3 occurrences de symboles de K sur sa droite, on va devoir avoir $2^3 = 8$ copies de la règle, pour prendre en compte toutes les combinaisons de "deviendra ϵ, deviendra pas ϵ". On a donc les règles

$E \to AaBEB$ *(déjà présent)*
$E \to \quad aBEB$
$E \to Aa \quad EB$
$E \to AaBE$
$E \to \quad a \quad EB$
$E \to \quad aBE$
$E \to Aa \quad E$
$E \to \quad a \quad E$

De même, on ajoute les règles $E \to Bb \mid Sb \mid b$.

Notre grammaire est donc maintenant

$$\begin{cases} S \to CD \mid AB \mid E \mid D \mid A \mid B \\ A \to a \mid BC \mid B \mid C \\ B \to abba \mid \cancel{\epsilon} \\ C \to BB \mid B \\ D \to Sa \mid a \\ E \to AaBEB \mid BSb \mid aBEB \mid AaEB \mid AaBE \mid aEB \mid aBE \mid AaE \mid aE \mid Sb \mid Bb \mid b \end{cases}$$

Il reste cependant une étape. En effet, La grammaire originelle engendrait notamment le mot vide, ce qui n'est plus le cas de la nouvelle. La troisième et dernière étape est est, si l'axiome S de la grammaire étudiée appartient à K, d'engendrer ϵ comme cas particulier. Pour ça, on ajoute un nouveau symbole S' qui remplace S comme axiome, ainsi que les règles $S' \to S \mid \epsilon$.

Exemple 5.4.9. *Au final, la version sans ϵ-production de la grammaire de l'exemple 5.4.7 est donc*

$$\begin{cases} S' \to S \mid \epsilon \\ S \to CD \mid AB \mid E \mid D \mid A \mid B \\ A \to a \mid BC \mid B \mid C \\ B \to abba \\ C \to BB \mid B \\ D \to Sa \mid a \\ E \to AaBEB \mid BSb \mid aBEB \mid AaEB \mid AaBE \mid aEB \mid aBE \mid AaE \mid aE \mid Sb \mid Bb \mid b \end{cases}$$

où S' est l'axiome.

Exercice 5.4.3. *Eliminer les ϵ-production de la grammaire suivante :*

$$\begin{cases} S \to ABC \mid BD \\ A \to BC \mid ab \\ B \to CDC \mid \epsilon \\ C \to BAB \mid ABA \\ D \to abba \mid BB \mid BCBS \end{cases}$$

Elimination de cycles

Si la grammaire est ϵ-libre, le seul moyen d'avoir des cyles est via des règles de la forme $A \to B$, ou **productions singulières**. Puisqu'on a un algorithme pour éliminer les ϵ-productions, on peut supposer que la grammaire étudiée est ϵ-libre. L'élimination de cyles peut donc être ramenée à l'élimination des productions singulières.

La première étape de l'algorithme consiste à trouver l'ensemble des paires de non-terminaux (A, B) telles que $A \to^* B$, cad l'ensemble des symboles qui peuvent se réécrire en un autre non-terminal. On procède encore une fois par point fixe sur les règles suivantes :

— Si $A \to B$, alors $A \to^+ B$
— Si $A \to^+ B$ et $B \to C$, alors $A \to^+ C$

Exemple 5.4.10. *Soit la grammaire*

$$\begin{cases} S \to B \mid CA \\ A \to C \mid a \\ B \to S \mid C \mid ab \\ C \to c \end{cases}$$

On obtient ces relations :

— $S \to^+ B$ *(règle 1)*
— $S \to^+ C$ *(règle 1 puis règle 2)*
— $A \to^+ C$ *(règle 1)*
— $B \to^+ S$ *(règle 1)*
— $B \to^+ C$ *(règle 1)*

Une fois qu'on a identifié ces paires (A, B), il suffit d'*inliner*, cad. de copier/coller les productions non-singulières de B dans celles de A. Ce faisant, on permet de faire immédiatement des réécritures qui se faisaient sinon en plusieurs étapes.

Exemple 5.4.11. *En continuant l'exemple 5.4.10, on obtient*

$$\begin{cases} S \to \cancel{B} \mid CA \mid ab \mid c \\ A \to \cancel{C} \mid a \mid c \\ B \to \cancel{S} \mid \cancel{C} \mid ab \mid CA \mid c \\ C \to c \end{cases}$$

Exercice 5.4.4. *Eliminer les cycles de la grammaire de l'exemple 5.4.5.*

Elimination de symboles inutiles

Chercher les symboles inaccessibles revient à calculer ceux qui sont accessibles, ce qui est plus simple et naturel. Pour ça, on part de toutes les règles ayant l'axiome dans leur partie gauche l'axiome, et tout symbole apparaîssant à droite d'au moins une règle est, de fait, accessible. On itére ensuite en partant à chaque fois des nouveaux symboles ajoutés, jusqu'à atteindre ... un point fixe !

Exemple 5.4.12. *On reprend la grammaire de l'exemple 5.4.6. S peut se réécrire immédiatement en A ou B, ces deux symboles sont donc accessibles depuis l'axiome. B peut uniquement se réécrire en ϵ, et ne permet donc pas d'accéder à de nouveaux symboles, au contraire de A. La règle $A \to CE$ nous fait en effet ajouter C et E à l'ensemble des symboles accessibles.*

C se réécrit uniquement en a et E en abbA. Puisque A faisait déjà partie de l'ensemble des états accessibles, on n'a rien à ajouter et le point fixe est atteint. Les états accessibles étant $\{S, A, B, Cs, E\}$, les inaccessibles en sont le complémentaire, soit $\{D, F\}$.

La grammaire se simplifie donc en

$$\begin{cases} S \to A \mid B \\ A \to CE \\ B \to \epsilon \\ C \to a \\ \sout{D \to Fa \mid \epsilon} \\ E \to abbA \\ \sout{F \to bD} \end{cases}$$

L'élimination des symboles sans production fonctionne de façon analogue, mais en partant de la fin. On va donc chercher à déterminer l'ensemble des symboles avec production, en commençant par repérer les symboles qui se réécrivent en un "vrai mot" (composé uniquement de terminaux). Ensuite, on itère.

Exemple 5.4.13. *On reprend à nouveau la grammaire (pas simplifiée, par souci pédagogique) de l'exemple 5.4.6. B, C et D peuvent se réécrire immédiatement en des "vrais mots" (ϵ pour B et D, a pour C). Puisque qu'on sait maintenant que ces 3 non-terminaux peuvent sont productifs, on cherche maintenant des symboles qui peuvent se réécrire en un mot composé de terminaux, B, C et/ou D.*

On peut donc ajouter à l'ensemble des symboles productifs S (qui se réécrit immédiatement en B) et F (qui se réécrit immédiatement en bD). Ces nouveaux ajouts ne rendent cependant ni A ni E productifs, on a donc atteint le point fixe.

Puisque les symboles productifs sont $\{S, B, C, D, F\}$, ceux sans production sont $\{A, E\}$. On peut donc simplifier la grammaire en

$$\begin{cases} S \to \sout{A} \mid B \\ \sout{A \to CE} \\ B \to \epsilon \\ C \to a \\ D \to Fa \mid \epsilon \\ \sout{E \to abbA} \\ F \to bD \end{cases}$$

Exercice 5.4.5. *Eliminer les symboles inutiles de la grammaire suivante :*

$$\begin{cases} S \to BAB \mid DB \\ A \to B \mid a \\ B \to bB \mid Ba \mid DAD \\ C \to c \mid AB \mid AA \\ D \to Da \mid B \mid aSb \end{cases}$$

Chapitre 6

Notion(s) de calculabilité

Les automates forment un langage de programmation, et donc une façon de penser et d'automatiser le calcul. On commence par introduire deux autres modèles bien connus, puis on discutera du contexte historique et scientifique ayant mené à ces notions.

Ce chapitre ne se substitue pas à une authentique introduction à la calculabilité, et devrait d'ailleurs au strict minimum être complété par la magnifique leçon inaugurale de Xavier Leroy au Collège de France. Il est très probable qu'il contienne des inexactitudes historiques, voire scientifiques. Plutôt que la précision ou l'exhaustivité, il a avant tout pour objectif de contextualiser ceux qui le précèdent, ainsi que de donner une idée, et pourquoi pas le goût, des questions qui se posent dans l'étude des modèles de calcul, ou **calculabilité**.

6.1 Différents modèles de calcul

L'informatique a pour vocation automatiser ce qui peut l'être afin de faire réaliser ces tâches par des machines plutôt que des humains. Une tâche est automatisable si elle peut être résolue par une série d'instructions sans ambiguïté, c'est-à-dire si elle peut être réduite à des calculs qu'il s'agit de définir formellement.

D'un point de vue programmation, 2 principaux modèles historiques co-existent : les **machines de Turing** et le **λ-calcul**. Ces langages, qui remontent tout de même aux années 30, ne sont pas utilisables en pratique (encore que), mais posent les fondamentaux de ce qu'est un langage de programmation.

Machines de Turing Les machines de Turing disposent de la même notion d'état que les automates, mais aussi d'une mémoire infinie modifiable. Cette mémoire, généralement appelée ruban, contient initialement le mot donné en entrée. Comme les automates, les transitions des machines de Turing sont déterminées par l'état actuel ainsi que la lettre lue. Cependant, en plus de potentiellement changer l'état, les transitions indiquent comment se déplacer dans le mot (droite comme dans les automate, mais aussi aller à gauche ou rester sur place) et une éventuelle réécriture. Les machines de Turing peuvent donc manipuler à loisir leur mémoire, quitte à réécrire le mot donné en entrée ou déborder d'un côté ou de l'autre. Cette manipulation explicite de la mémoire, ainsi que le fait qu'elles favorisent l'utilisation de boucles, rapprochent fortement les machines de Turing de la programmation impérative (C, langage machine, le coeur de Python

etc). Elles ont été conçues par l'anglais **Alan Turing**.

λ-calcul A la différence des machines de Turing, qui ont une approche quasiment mécanique (pour ne pas dire "bidouille") de l'exécution d'un programme, le λ-calcul est profondément mathématique et repose sur la récursion. Tout n'y est que fonction, au point que ces dernières sont des objets comme les autres, notamment passables en arguments. On citera l'exemple classique d'une fonction qui reçoit une fonction de tri et une liste, et renvoie la liste triée selon la fonction fournie. Le λ-calcul est la base de la programmation fonctionnelle. Il a été crée par **Alonzo Church**.

Remarque Les types en programmation impérative n'ont souvent qu'une valeur de garde-fou contre des opérations totalement absurdes, alors qu'ils ont une fonction beaucoup plus structurante (certain.e.s diraient "contraignante") en programmation fonctionnelle. L'utilisation de fonctions comme arguments oblige par exemple à repenser les types et aller plus loin que les classiques `bool`, `int` et cie. On renverra encore une fois à la présentation de Xavier Leroy citée en introduction pour une meilleur vision d'ensemble.

Ces deux modèles ne forment pas l'alpha et l'omega de la calculabilité, qui contient de nombreux modèles plus ou moins exotiques, comme les fonctions μ-récursives ou les automates cellulaires (voir à ce sujet la très chouette vidéo de la chaîne ScienceEtonnante).

6.2 Un peu d'Histoire : le programme de Hilbert

Bien que développés en isolation, ces deux modèles s'inscrivent dans un même mouvement intellectuel. A l'aube du 20^{eme} siècle, les fondements des mathématiques sont mis à mal (pour ne pas dire balayés) par plusieurs paradoxes, dont le plus connu est le paradoxe de Russel. Le mathématicien allemand David Hilbert pose alors les bases d'un grand plan de (re)fondation des mathématiques, aujourd'hui appelé "programme de Hilbert". L'idée était d'obtenir une formalisation de l'intégralité des mathématiques, formalisation qui se devait d'être **complète**, **cohérente** et **décidable**.

6.2.1 Complétude et cohérence

Un système mathématique est dit **complet** s'il existe un système de règles précis dans lequel tout énoncé qui y est vrai peut être démontré. Par exemple, la logique du premier ordre admet plusieurs systèmes de preuve (déduction naturelle, calcul des séquents, ...), contenant des règles du type "De $A \wedge B$ on peut déduire A" ou "Si on a un prédicat $P(x)$ et une constante a telle que $P(a)$ est vrai, alors on peut déduire $\exists x, P(x)$". Le mathématicien austro-hongrois Kurt Gödel montre en 1929 que la logique du premier ordre est complète.

Il existe donc un système de déduction entièrement formalisé et syntaxique dans lequel on peut démontrer tout énoncé vrai de logique du premier ordre (par exemple $(\exists y, \forall x, P(x, y)) \rightarrow \forall x, \exists y, P(x, y)$). En ce sens, le **théorème de complétude de Gödel** relie la sémantique (la vérité ou non) et la syntaxe (la déduction par application de règles) de la logique du premier ordre.

Ce résultat, avec d'autres, va dans le sens de Hilbert. La logique du premier ordre est cependant trop faible pour exprimer l'arithmétique, qui constitue à l'époque le graal des mathématiques. Cette réalité sera douloureusement rappelée l'année suivante, en 1930, par Gödel, lorsqu'il présentera son *magnum opus*, appelé aujourd'hui **théorème d'incomplétude de Gödel**. Ce théorème

établit que n'importe quel système formel assez puissant pour exprimer les nombres entiers sera incomplet. Quelle que soit la formalisation de l'arithmétique choisie, il y aura donc toujours au moins un résultat vrai qui n'y sera pas prouvable. La preuve de Gödel est très bien vulgarisée dans cette vidéo.

Un système mathématique est dit **cohérent** si on ne peut pas y prouver une chose et son contraire (ou, de façon équivalente, si on ne peut y prouver l'énoncé "faux"). Sans aller jusqu'à casser la cohérence de quel que système mathématique que ce soit, la preuve du théorème d'incomplètude de Gödel a pour corollaire que toute formalisation de l'arithmétique sera incapable de prouver sa propre cohérence - ce qui ne veut pas dire qu'elle ne l'est pas.

Les travaux de Gödel n'enterrent pas le programme de Hilbert, mais lui portent un sacré coup. Il faut en effet revoir à la baisse les ambitions, et arrêter d'espérer une formalisation "absolue" des mathématiques. Dans une de ces cruelles ironies dont l'Histoire a le secret, Gödel a présenté ses travaux lors d'une conférence organisée pour le départ en retraite d'Hilbert.

6.2.2 Décidabilité et thèse de Church

Au 17^{eme} siècle, Leibniz rêve d'une procédure permettant de déterminer automatiquement, *via* un calcul, si une formule mathématique est vraie ou non. Leibniz se rendit compte que les bases formelles n'étaient alors pas disponibles, notamment la formalisation du calcul. Hilbert relance et précise cette ambition en espérant un système qui serait **décidable**, cad. dont la vérité de tout énoncé pourrait être déterminée par une série finie et établie de calculs. Dit autrement, Hilbert prévoyait l'existence d'un algorithme qui prendrait en entrée toute formule de sa formalisation, et renverrait "vrai" ssi. la formule est effectivement vraie. Ce problème portait alors le doux nom d'**Entscheidungsproblem** (problème de la décision).

Toujours dans l'optique d'avoir une formalisation de bout en bout des mathématiques, la question de la décidabilité appelle à une définition précise de la notion de calcul. C'est dans ce cadre que, en 1936, Church propose le λ-calcul, dont il montre immédiatement qu'il est indécidable, et que Turing présente ses machine, ainsi qu'également une preuve de leur indécidabilité. L'existence de deux formalisations indécidables du calcul n'est en soi pas un problème, tant qu'il en existe une décidable.

Cependant, Church et Turing ont rapidement montré que leurs modèles sont équivalents, dans le sens où toute fonction exprimable dans un modèle le sera dans l'autre. On dit que les modèles ont la même **expressivité**. Certaines fonctions très simples en λ-calcul seront un enfer à coder en machine de Turing, et inversement [1], mais il reste pour le moins étonnant que deux modèles fonctionnant de façons si orthogonales aient, au fond, la même puissance.

De nombreux autres modèles seront également montrés équivalents, ce qui pose la question d'une notion "naturelle" et indépassable de calcul, hypothèse appelée **thèse de Church**. On ne s'attardera pas sur les problématiques épistémologiques ou philosophiques liées, qui sont introduites dans [1], sinon en disant que le programme de Hilbert s'en retrouve encore une fois mis à mal. Les preuves d'équivalence reposant en effet sur des traductions d'un modèle à un autre (il s'agit donc de preuves constructives), ce qui implique que l'indécidabilité se "transmet" entre modèles.

La classe des modèles équivalents aux machines de Turing forme l'ensemble des modèles dits **Turing-complets**. En plus des machines de Turing et le λ-calcul, on y retrouve les automates cellulaires, les fonctions μ-récursives, l'immense majorité des langages de programmation (Py-

1. Penser à la différence entre compétence et performance.

thon, C, OCaml, Java, et même Makefile, Bash ou LaTeX sont bel et bien aussi expressifs les uns que les autres) ou, de façon plus surprenante, Minecraft et powerpoint.

Il est en fait étonnement compliqué de créer un système qui soit non-trivial sans pour autant être Turing-complet (on y revient en 6.3), cad. sans être en fait un langage de programmation "normal".

Pour le plaisir, on revient sur les preuves d'indécidabilité mentionnées plus haut. Puisque les machines de Turing, le λ-calcul et les langages de programmations "classiques" sont équivalents, on va utiliser des notations plus simples et modernes pour présenter d'abord une preuve de l'indécidabilité d'un problème précis :

Théorème 13. *(Indécidabilité du problème de l'arrêt) Savoir si un programme termine est un problème indécidable.*

Démonstration. On va procéder par l'absurde (cf. annexe A.1.1). La décidabilité du problème de l'arrêt signifierait qu'il existe un programme, appelé A, qui prend en argument un programme P et un élément x, et renvoie `true` si et seulement si $P(x)$ termine.

A partir de A, on peut constuire un autre programme appelé B, qui prend en argument un programme P, et termine si et seulement si $P(P)$ ne termine pas :
`def B P := if (A P P) then (while true skip) else skip`.

Maintenant, appliquons B à lui même. On utilisant la définition de B, on obtient
`B(B) := if (A B B) then (while true skip) else skip`, ce qui veut dire que $B(B)$ termine si et seulement si $B(B)$ ne termine pas. On obtient donc un paradoxe, signifiant que notre seule hypothèse, l'existence de A, est fausse. □

Remarque La preuve contient une bizarrerie, à savoir l'application d'un programme à lui-même ($P(P)$ et $B(B)$). Une telle chose est proscrite par les types, dont l'utilisation rendrait la preuve donnée caduque. Il est cependant toujours possible de prouver l'indécidabilité de l'arrêt pour des programmes typés, de façon cependant plus tordue.

Au-delà de ce problème particulièrement connu, il existe une véritable armée de problèmes indécidables, comme spécifié par le théorème de Rice :

Théorème 14. *(Théorème de Rice) On appelle propriété sémantique non-triviale une propriété sur le comportement d'un programme telle qu'il existe au moins un exemple la respectant et un ne la respectant pas. Toute propriété sémantique non-triviale est indécidable.*

Démonstration. On procède encore une fois par l'absurde, en supposant qu'il existe une propriété sémantique non-triviale i décidable. Puisque i est non-triviale, on sait qu'il existe P_{i+} (resp. P_{i-}) un programme qui satisfait (resp. ne satisfait pas) la propriété i. On va montrer qu'il est alors possible de résoudre le problème de l'arrêt.

Soit un programme P dont on veut vérifier qu'il termine sur l'argument x. On vérifie d'abord si P satisfait la propriété i. Supposons, sans perte de généralité (il suffit sinon d'inverser les $+$ et $-$), que ce n'est pas le cas. On écrit alors un programme qui fait tourner $P(x)$, puis P_{i+}. On vérifie si le tout satisfait i. Si c'est le cas, on sait que $P(x)$ a fini. A l'inverse, si ce n'est pas le cas, P_{i+} n'a pas été atteint, ce qui veut dire que $P(x)$ n'a pas fini.

L'existence supposée de la décidabilité propriété sémantique non-triviale permet de résoudre le problème de l'arrêt, pourtant indécidable. Il n'existe donc pas de telle propriété. □

Remarque Une telle preuve, où on montre que la décidabilité d'un problème P_1 permettrait de résoudre un problème P_2 pourtant indécidable, s'appelle une preuve par réduction, puisqu'on y réduit la décidabilité de P_2 à celle de P_1.

Si ces recherches et résultats remettent encore une fois en question le programme de Hilbert tel qu'il fût initialement pensé et formulé, y voir un échec serait quelque peu pessimiste. En effet, si on a *perdu* un modèle absolu des mathématiques - qui ont tout de même obtenu des fondations plus stables dans toute cette affaire, on y a gagné l'automatisation, la mise en oeuvre, le *déploiement* des maths, qu'on appelle aujourd'hui **informatique**. Si certaines questions ou bases ont plusieurs siècles (on conseillera à ce sujet la lecture de [2], ainsi que la visite du Musée des arts et métiers à Paris ou du *Computer History Museum* en Californie), les progrès de l'informatique tout au long du 20^{eme} siècle sont absolument phénoménaux, et le sont restés à date d'écriture.

6.3 Et les automates alors ?

Intuitivement, les propriétés sémantiques sont indécidables sur tout modèle Turing-complet car ces derniers sont trop puissants, ou expressifs. Dans certains cas, on est prêts à sacrifier un peu d'expressivité en échange de plus de décidabilité. Commence alors un jeu consistant à affaiblir les modèles pour qu'on puisse décider des propriétés à leur sujet, sans pour autant qu'ils en deviennent trivial. La hierarchie de Chomsky, étudiée en 5.3, en est un bon exemple.

6.3.1 Type 3 et automates finis

On a déjà vu que les grammaires de type 3 sont équivalentes aux automates finis. Ces derniers sont tellement faibles que de nombreuses (l'intégralité ?) des propriétés sémantiques y sont décidables. En particulier, la terminaison est garantie par construction (une opération par lettre du mot en entrée).¨Pour ce qui est de l'équivalence entre deux automates finis, on commence par les déterminiser et minimiser. Si les deux automates obtenus ont le même nombre d'état, il s'agit de trouver les paires d'états qui se correspondent (c'est ce qu'on appelle une **bisimulation**). De nombreux algorithmes plus ou moins efficaces existent, mais le nombre d'états et donc de possibilités étant fini, on peut fondamentalement tout tester en un temps lui-même fini.

6.3.2 Type 2 et automates à pile

Les grammaires de type 2 sont équivalentes aux **automates à pile** (*pushdown automata*). Il s'agit d'automates finis étendus avec une pile, et dans lesquels les transitions ou acceptations dépendent non seulement de l'état actuel, mais aussi du symbole au sommet de la pile. Par exemple, le langage $\{a^n b^n \mid n \in \mathbf{N}\}$ sera accepté par un automate à deux états. On boucle sur le premier en enpilant un symbole à chaque lecture de a. La lecture d'un b fait passer sur le second état, sur lequel une boucle en b enlèvele symbole au sommet de la pile. On accepte ssi. la pile est vide.

Comme pour les automates finis, la terminaison des automates à pile est garantie par construction. La décidabilité de l'équivalence entre automates à pile déterministes a été montrée en 1997 par Géraud Sénizergues (qui a remporté le prix Turing pour cette découverte), tandis que l'équivalence entre automates à pile non-déterministes a été montrée indécidable. Enfin, il a été montré qu'un "automate à deux piles" est Turing-complet (moralement, on peut simuler le ruban infinie d'une machine de Turing à l'aide de deux piles).

6.3.3 Type 1 et automates linéairement bornés

Les grammaires de type 1 sont quant à elles équivalentes aux **automates linéairement bornés**. Il s'agit de machines de Turing, au détail près que le ruban est borné, linéairement pour chaque entrée. Par exemple, pour un problème donné, on s'accordera une mémoire de $3 \times l + 28$ cellules, où l est la longueur du mot donné en entrée. L'arrêt d'un automate linéairement borné est décidable, contrairement à l'équivalence, l'*emptiness* (le fait de déterminer, pour un automate donné, si le langage qu'il reconnaît est vide) ou la finitude (déterminer si l'automate reconnaît un nombre fini de mots).

6.3.4 Type 0 et machines de Turing

Comme on pouvait s'y attendre en grimpant des grammaires de type 3 à celles de type 0, cette dernière catégorie est équivalente aux machines de Turing, et donc à tout modèle Turing-complet.

La hierarchie de Chomsky est bien loin de constituer l'intégralité des classes de modèles de calculs (et on ne mentionne même pas les quelques classes de complexité auxquelles associer ces problèmes), mais permet de découvrir un amusant premier dégradé. Bien qu'étant la source de l'Informatique, la recherche en calculabilité est encore aujourd'hui très riche, tant en modèles et problèmes abscons, ainsi qu'en questions les concernants.

Bibliographie

[1] G. Dowek. *Les métamorphoses du calcul : une étonnante histoire de mathématiques.* Le Pommier, 2007.

[2] Jean-Noël Lafargue et Marion Montaigne. *L'intelligence artificielle.* Les éditions du Lombard, 2016.

Annexe A

Rappels mathématiques

A.1 Logique

A.1.1 Raisonnement par l'absurde

Définition A.1.1: Raisonnement par l'absurde

Un **raisonnement par l'absurde** consiste à prouver une chose en 1) supposant son contraire et 2) montrer que ça fout tout en l'air. Plus formellement, pour prouver P, on suppose $\neg P$ et on montre que ça nous permet de déduire $\bot$, ce qui veut dire soit que la logique est incohérente, soit que $\neg P$ est fausse, et donc que P est vraie.

Exemple A.1.1. *Imaginons une situation où les rues sont sèches, et où on voudrait prouver qu'il n'a pas plu. On suppose alors l'inverse, c'est-à-dire qu'il a plu. Or, s'il a plu, les routes sont mouillées. On obtient alors que 1) les routes sont mouillées et 2) les routes ne sont pas mouillées, ce qui est un paradoxe. La seule hypothèse faite étant le fait qu'il a plu, elle doit être fausse.*

Exemple A.1.2. *On veut prouver qu'il existe une infinité de nombres premiers. On suppose l'inverse, cad. qu'il y en a un ensemble fini* $\{p_1, ..., p_n\}$. *Soit* $n = 1 + \prod_{i \in [1-n]} p_i = 1 + p_1 \times ... \times p_n$. *$n$, comme tout nombre, admet au moins un diviseur premier.*

Or, n est strictement plus grand que tout nombre premier et ne peut donc pas en être un. De plus, pour tout $i \in [1-n]$, $\frac{n}{p_i} = p_1 \times ... \times p_{i-1} \times p_{i+1} \times ... \times p_n + \frac{1}{p_i}$. *Tout nombre premier étant* ≥ 2, $\frac{1}{p_i}$ *ne forme pas un entier, et donc* $\frac{n}{p_i}$ *non plus.*

On obtient une contradiction, notre hypothèse sur la finitude des nombres premiers est donc fausse.

A.2 Ensembles

A.2.1 Opérations entre ensembles

On suppose dans cette section deux ensembles E_1 et E_2 d'éléments de même nature. De plus, on rappelle que $\mathbb{N}$ est l'ensemble des entiers naturels (0, 1, 2, ...) et on pose $P = \{x \in \mathbb{N} \mid x \text{ pair}\}$ et $I = \{x \in \mathbb{N} \mid x \text{ impair}\}$.

Définition A.2.1: Union

L'union de E_1 et E_2, notée $E_1 \cup E_2$, est l'ensemble des éléments appartenant à E_1 ou E_2 (ou les deux). Plus formellement, $E_1 \cup E_2 = \{x \mid x \in E_1 \vee x \in E_2\}$.

Exemple A.2.1. $P \cup I = \mathbb{N}$.

Définition A.2.2: Intersection

L'intersection de E_1 et E_2, notée $E_1 \cap E_2$, est l'ensemble des éléments appartenant à E_1 et E_2. Plus formellement, $E_1 \cap E_2 = \{x \mid x \in E_1 \wedge x \in E_2\}$.

Exemple A.2.2. $P \cap I = \emptyset$.

Définition A.2.3: Différence

La différence (ensembliste) de E_1 et E_2, notée $E_1 \backslash E_2$, est l'ensemble des éléments appartenant à E_1 mais pas à E_2. Plus formellement, $E_1 \backslash E_2 = \{x \mid x \in E_1 \wedge x \notin E_2\}$.

Exemple A.2.3. $\mathbb{N} \backslash I = P$ *et* $\mathbb{N} \backslash P = I$.

Exemple A.2.4. $I \backslash \mathbb{N} = \emptyset$.

Exemple A.2.5. $E_1 \backslash \emptyset = E_1$.

Définition A.2.4: Complémentaire

Le complémentaire de E_1, noté $\overline{E_1}$, est l'ensemble des élément (de même nature) n'appartenant pas à E_1.

Exemple A.2.6. $\overline{P} = I$ *et* $\overline{I} = P$.

Remarque Soit E l'ensemble des éléments du même type que ceux de E_1 (par exemple $\mathbb{N}$ pour P ou I), alors $\overline{E_1} = E \backslash E_1$.

Définition A.2.5: Produit d'ensembles

Soit deux ensembles E_1 et E_2, contenant respectivement des éléments de types τ_1 et τ_2. Soit également $\cdot$ une opération de type $\tau_1 \to \tau_2 \to \tau_3$, cad. une opération qui prend en argument gauche un élément de type τ_1 et à droite un argument de type τ_2 et renvoie un objet de type τ_3, alors

$$E_1 \cdot E_2 = \{x \cdot y \mid x \in E_1 \ \wedge \ y \in E_2\}$$

Dit autrement, un produit d'ensembles renvoie l'ensemble des combinaisons d'éléments de deux ensembles *via* une opération fournie. Si l'opération $\cdot$ est un endormorphisme, cad. qu'elle est de type $\tau \to \tau \to \tau$, alors on peut itérer le produit de la façon suivante :

$$E^0 = \{1\} \qquad \text{Où 1 est l'élément neutre de } \tau$$
$$E^{n+1} = E^n \cdot E$$

Cette notion très générale ne doit pas être confondue avec

Définition A.2.6: Produit cartésien

Soit deux ensembles E_1 et E_2, ne contenant pas forcément des éléments de même type, alors

$$E_1 \times E_2 = \{(x, y) \mid x \in E_1 \ \wedge \ y \in E_2\}$$

Le produit cartésien, noté $\times$, renvoie l'ensemble des couples d'éléments de deux ensembles donnés. Il s'agit d'un cas particulier du produit d'ensemble, où l'opération est la "mise en couple". Cette opération ne pouvant pas être un endormorphisme, le produit cartésien ne peut être itéré.

A.2.2 Ensemble des parties

Soit un ensemble X, on note $P(X)$ (parfois 2^X) l'ensemble de ses parties (ou *powerset*), cad l'ensemble des ensembles formés à partir d'éléments de X.

Exemple A.2.7. *Soit $X = \{x, y, z\}$, alors - en classant les éléments par leur cardinal -*

$$\begin{aligned} P(X) = \{ \quad & \emptyset, \\ & \{x\}, \quad \{y\}, \quad \{z\}, \\ & \{x, y\}, \quad \{x, z\}, \quad \{y, z\} \\ & \{x, y, z\} \qquad\qquad \}. \end{aligned}$$

Lemme 11. $\forall X, \emptyset \in P(X) \wedge X \in P(X)$.

Lemme 12. $\forall X, |P(X)| = 2^{|X|}$.

Démonstration. Pour générer l'ensemble des sous-ensembles de X, on choisit de prendre ou non chaque élément de X. On a donc $\underbrace{2 \times 2 \times ... \times 2}_{\text{n fois}}$ choix, d'où les $2^{|X|}$ au total. □

A.3 Algorithmique

A.3.1 Itératif vs. récursif, le cas des parcours d'arbre

On va comparer les implémentations itératives et récursives d'un parcours d'arbre. On regardera, sans perte de généralité, un parcours préfixe qui effectue une opération f (par exemple afficher, peu importe) sur tous les éléments de l'arbre.

Commençons par l'itératif. Puisqu'on a (littéralement) des branchements, contrairement à une liste où on pourrait juste foncer tout droit, on va avoir besoin d'une mémoire. L'idée va être d'avoir une *todo-list*, sous forme de pile, et d'explorer l'arbre en la suivant. On commence avec seulement la racine dans la pile et, chaque fois qu'en extrait un noeud comme dans la figure A.1, exécuter $f(x)$ et se noter qu'on doit explorer les arbre d_1, d_2 et d_3. Au total, on a l'algorithme de la figure A.2.

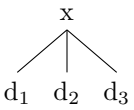


FIGURE A.1 – Exemple de noeud

```
# fonction tree_it(arbre)
todo = stack()
todo.add(root(arbre))
while (todo pas vide):
    current = todo.pop()
    if current is Leaf(x):
        f(x)
    // si current est un noeud interne avec
    // un element x et la liste de descendants descs
    elif current is Noeud(x,descs):
        f(x)
        // pour du prefixe, on doit avoir
        // d1 plus haut dans la pile que d2
        for d in reverse(descs):
            todo.add(d)
```

FIGURE A.2 – Parcours itératif d'arbre

Il n'est pas totalement trivial de se convaincre que cet algorithme visite bien tous les noeuds d'un arbre donné dans un ordre préfixe, notamment en comparaison de la version récursive trouvable en figure A.3

```
# fonction tree_rec(arbre)
if arbre is Leaf(x):
    f(x)
elif arbre is Noeud(x,descs):
    f(x)
    for d in descs:
        tree_rec(d)
```

FIGURE A.3 – Parcours récursif d'arbre

La version récursive est dite **de haut-niveau**, dans le sens où elle est très proche de la définition d'un parcours d'arbre, alors que la version itérative est plus **de bas-niveau**, en ce qu'elle implémente cette définition. Dit autrement, la version récursive est plus abstraite, là où la version itérative est plus concrète, dans la mise en pratique de la définition. D'ailleurs, la figure A.2 correspond bien à l'exécution de la figure A.3, où la pile *todo* remplace la pile d'appels, qui gère les appels récursifs en suivant "où on en est" et ce qu'il reste à faire.

Ces deux approches ont bien sûr leurs avantages et inconvénients. La programmation de haut-niveau permet de se convaincre - voire de prouver - bien plus facilement qu'un programme réalise ce qu'il est censé faire, et est plus facilement lisible / réutilisable. Par contre, jouer soi-même avec l'implémentation de la récursion permet de gérer à la main ces mécanismes et donc potentiellement d'optimiser tout ça - bien que les compilateurs appliquent des optimisations de plus en plus puissantes, bien souvent plus sûres et malignes que ce qu'on peut imaginer.