Bases formelles du TAL

Pierre-Léo Bégay

3 février 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

Table des matières

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*
- *abcabcacbacbacbacbabcabcabcabcabcabcabc*
- *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

$$
\begin{array}{lll}
 & |w_1.w_2| & \\
= & |c.w_1'.w_2| & \text{par définition de } w_1 \\
= & 1 + |w_1'.w_2| & \text{par définition de } |.| \\
= & 1 + (|w_1'| + |w_2|) & \text{par hypothèse d'induction} \\
= & (1 + |w_1'|) + |w_2| & \text{par associativité de l'addition} \\
= & |c.w_1'| + |w_2| & \text{par définition de } |.| \\
= & |w_1| + |w_2| & \text{par définition de } w_1
\end{array}
$$

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.2.2.

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.

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

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

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

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

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* $\{w \in \Sigma^* \mid |w| \ pair\}$.

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") ?*

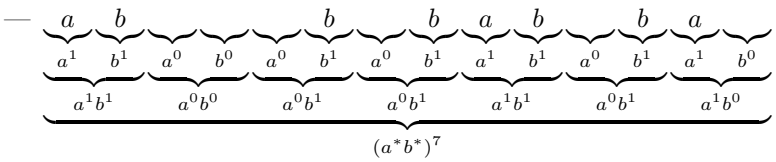
On peut même faire encore plus rigolo, en enchâssant les étoiles :

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

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

— $\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{}_{a^0}\underbrace{bb}_{b^2}}_{a^0b^2}\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{}_{a^0}\underbrace{b}_{b^1}}_{a^0b^1}\underbrace{\underbrace{a}_{a^1}\underbrace{}_{b^0}}_{a^1b^0}}_{(a^*b^*)^5}$



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*

$$\underbrace{\underbrace{\underbrace{bb\ bb\ bb\ bb}_{(bb)^4}}_{(bb)^*}}_{(aa)^* + (bb)^*}$$

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*

$$\underbrace{\underbrace{bb}_{aa+bb}\ \underbrace{aa}_{aa+bb}\ \underbrace{bb}_{aa+bb}\ \underbrace{bb}_{aa+bb}\ \underbrace{bb}_{aa+bb}\ \underbrace{aa}_{aa+bb}}_{(aa+bb)^6}$$

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

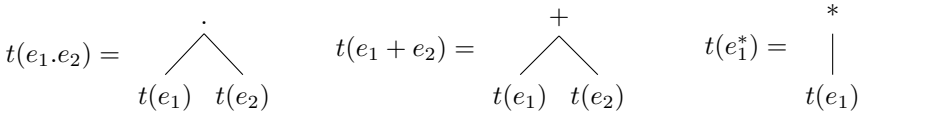
La figure 2.1 se lit "une expression rationnelle e est

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*

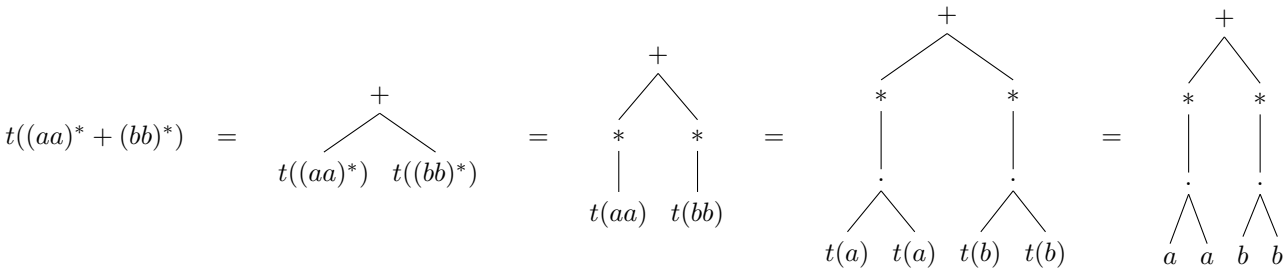


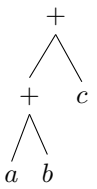
FIGURE 2.4 – Analyse syntaxique de $(aa)^* + (bb)^*$

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.

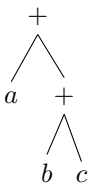
Comme on le verra dans la partie sémantique, les symboles $+$ et $.$ sont **associatifs**, ce qui veut dire que, pour toutes expressions e_1, e_2 et e_3, $(e_1 + e_2) + e_3$ et $e_1 + (e_2 + e_3)$ ont le même sens [3], et pareil avec la concaténation. Malgré une méfiance justifiée des arbres ternaires 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

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

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 $a + b + c$

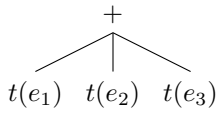
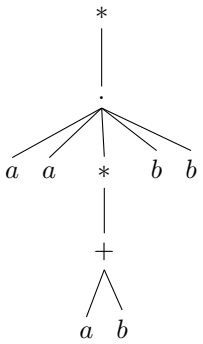


FIGURE 2.6 – Ambiguïté syntaxique assumée de $a + b + c$



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.2.2). 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*

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

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

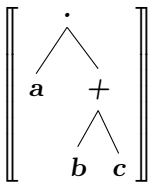
$$\left[\!\left[\begin{array}{c} + \\ \diagup\ \diagdown \\ e_1 \quad e_2 \end{array} \right]\!\right] = [\![e_1]\!] \cup [\![e_2]\!]$$

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

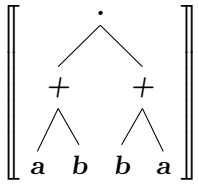
à *parser* automatiquement toute expression rationnelle qu'il ou elle lit. Ne vous y trompez pas cependant : l'analyse sémantique s'opère bien sur un arbre plutôt que sur une expression "plate".

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.

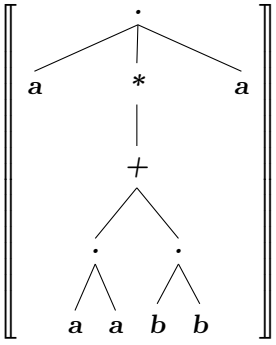
2.3.4 Sémantique de l'itération

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*



$$\begin{aligned}
&= [\![a]\!].[\![(aa+bb)^*]\!].[\![a]\!] \\
&= \{a\}.\textstyle\bigcup_{n\in\mathbf{N}}[\![aa+bb]\!]^n.\{a\} \\
&= \{a\}.\textstyle\bigcup_{n\in\mathbf{N}}\{aa,bb\}^n.\{a\} \\
&= \{a.\epsilon.a, a.aa.a, a.bb.a, a.((aa).(aa)).a, a.((aa).(bb)).a, a.((bb).(aa)).a, a.((bb).(bb)).a, ...\} \\
&= \{aa, aaaa, abba, aaaaaa, aaabba, abbaaa, abbbba, ...\}
\end{aligned}$$

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 *baabbaaabaababbbaaa*). 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 (3.4).

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 → 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

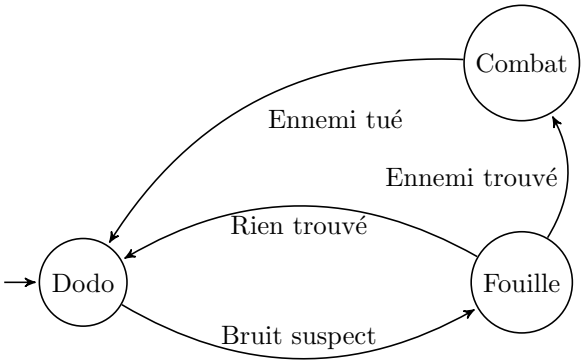


FIGURE 3.1 – Comportement des gardes d'un jeu imaginaire

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").*

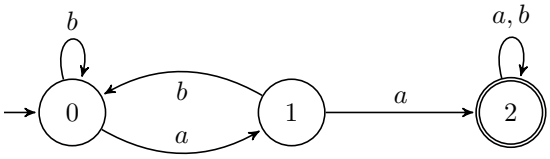


FIGURE 3.2 – Un premier automate

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.*

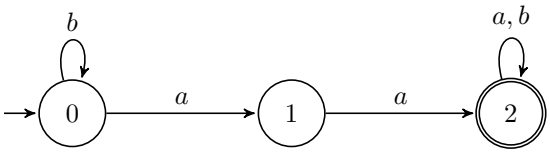


FIGURE 3.3 – Un automate incomplet

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]*.*

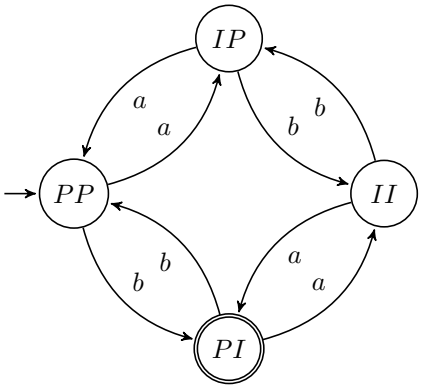
Il n'est pas question de compter les a et les b comme on pourrait naïvement l'imaginer, non seulement puisqu'il faut se contenter d'un nombre fini d'états, mais aussi parce que c'est beaucoup plus d'information que nécessaire. Les seules données qui nous intéressent sont en effet la parité du nombre 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 autuomate 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

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.

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

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

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 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.

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 $[f(a_1), ..., f(a_n)]$.

3.2 Automates finis non-déterministes

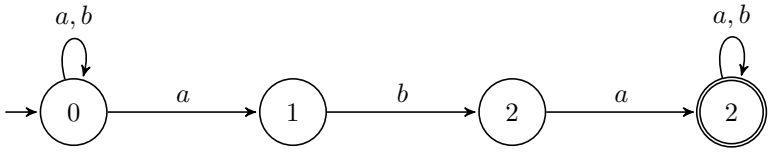
Les automates finis qu'on a vus jusqu'ici sont déterministes, en ce qu'un automate admet au maximum une seule transition par lettre, et donc que tout mot n'a lui-même pas plus d'un chemin. On va ici voir une nouvelle classe d'automates, les automates finis non-déterministes (AFND, ou *NDFA* pour *non-deterministic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

3.2.1 Principe général

Le non-déterminisme se manifeste de deux façons. On a d'abord les ϵ-transitions, qui seront étudiées à part dans le cadre d'un DM. Ici, on se concentrera sur la possibilité d'avoir plusieurs transitions pour le même couple état $\times$ lettre, ainsi que plusieurs états initiaux.

On peut donc 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 :

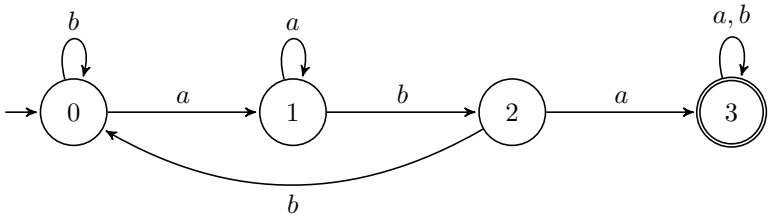
- $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{b} 0$
- $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 1 \xrightarrow{b} 2$
- $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 3$

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

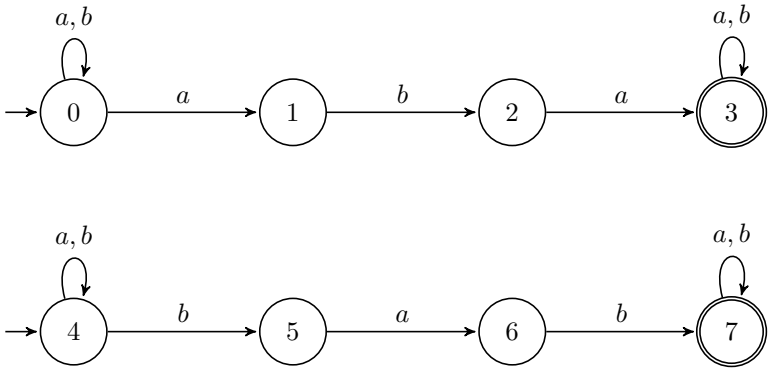
Un tel automate a l'avantage de reconnaître très clairement le langage $\Sigma^* aba \Sigma^*$. On peut par exemple le comparer à sa version déterministe qui, malgré la simplicité du langage, perd déjà pas mal en lisibilité :

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 au-



tomates 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.

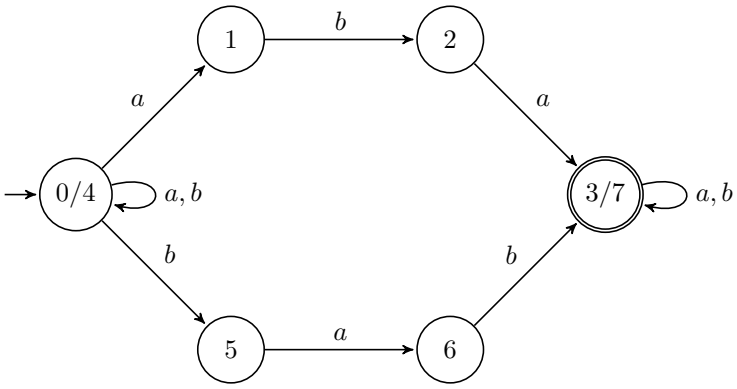
Exemple 3.2.2. *On veut reconnaître le langage des mots qui contiennent le facteur aba ou (inclusif) bab. On peut exploiter la possibilité d'avoir plusieurs états initiaux, et tout simplement produire l'automate suivant :*



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 $L = \{w \mid aba \text{ est un sous-mot de } w\}$.*

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 $(Q \times \Sigma^*) \to P(Q)$:

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

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.2.1.

```
# 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 premi re 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

3.3.1 Complétion

3.3.2 Déterminisation

3.3.3 Minimisation

3.4 Propriétés de clôture

3.4.1 Union

3.4.2 Intersection

3.4.3 Concaténation

3.4.4 Itération

Chapitre 4

Grammaires formelles

TODO

Chapitre 5

Introduction à la calculabilité

Chapitre 6

Théorème de Kleene et hiérarchie de Chomsky

TODO

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

$$P(X) = \{ \quad \emptyset, \quad \{x\}, \quad \{y\}, \quad \{z\}, \quad \{x, y\}, \quad \{x, z\}, \quad \{y, z\} \quad \{x, y, z\} \quad \}.$$

Lemme 7. $\forall X, \emptyset \in P(X) \land X \in P(X)$.

Lemme 8. $\forall X, |P(X)| = 2^{|X|}$.

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.2.2 Opérations entre ensembles

Définition A.2.1: 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

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

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.2.2: Produit cartésien

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

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

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.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.

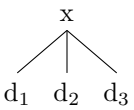


FIGURE A.1 – Exemple de noeud

```
# 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

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

```
# 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

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.