Bases formelles du TAL

Pierre-Léo Bégay

January 25, 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

Contents

Chapter 1

Notion de calculabilité

On s'attache d'abord à contextualiser le reste du cours en évoquant le cadre plus large de la calculabilité et les problématiques qui méneront à la notion d'automate. Ce chapitre peut être complété par la leçon inaugurale de Xavier Leroy au Collège de France.

1.1 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 d'une notion d'état, et d'une mémoire infinie modifiable. La notion d'état sera discutée en longueur dans la section 4, mais correspond en gros à une mémoire spéciale, propre à chaque programme, qui peut être modifiée et consultée facilement, notamment pour s'orienter dans le *flow* du programme[1]. Ces traits rapprochement fortement les machines de Turing de la programmation impérative (C, langage machine, le coeur de Python etc). Les machines de Turing sont dues à **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. 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**.

[1] Pensez à une variable booleene "premierefois" que vous avez sans doute déjà utilisée dans un `if` pour accéder ou non à un cas particulier

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 vidéo de la chaîne ScienceEtonnante).

Thèse de Church (ou thèse de Church-Turing) Church et Turing ont montré, dans les années 30, que les machines de Turing et le λ-calcul 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**. Attention cependant, certaines fonctions très simples en λ-calcul seront un enfer à coder en machine de Turing, et inversement[2]. Mais il reste remarquable que deux modèles fonctionnant de façons si orthogonales aient, au fond, la même puissance.

Ce résultat est moins surprenant avec le recul, puisqu'on sait maintenant que tous la plupart des modèles de calcul non-triviaux sont équivalents, et forment la classe des modèles **Turing-complets**. Il est en effet possible de simuler, ou coder, les machines de Turing ou le λ-calcul dans les fonctions μ-récursives ou les automates cellulaires, et inversement. Ce résultat s'étend à une armée de modèles, auxquels ils faut ajouter aujourd'hui des milliers de langages de programmation : Python, C, OCaml, Java, et même Makefile, Bash ou LaTeX sont bel et bien aussi expressifs les uns que les autres.

La thèse de Church peut même s'étendre à l'épistémologie ou à la philosophie, puisqu'elle peut sembler suggérer l'existence d'une notion "naturelle" et indépassable de calcul. On conseillera la lecture de [**?**] pour une introduction à ces problématiques.

Le λ-calcul est utilisé en NLP, mais sera étudié dans un autre cours. Les machines de Turing quant à elles n'ont, en soi, pas d'intérêt pour la linguistique, mais on peut les affaiblir pour les rendre paradoxalement plus pertinentes.

1.2 Décidabilité

Au $XVII^{ème}$ 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. Le problème réapparaît dans un cadre plus favorable, en 1928, lorsque Hilbert repose la question dans le cadre de son fameux programme de refondation des mathématiques. Le problème de la décision prend alors le doux nom d'**Entscheidungsproblem**.

La réponse ne se fera pas (trop) attendre, et c'est un double non. Church et Turing publient en 1936, mais indépendament, des preuves qu'une telle procèdure, ou plutôt un tel programme, est impossible à écrire en machine de Turing et λ-calcul, et donc pour tout modèle de calcul équivalent. Le problème de la décision est **indécidable**, et il est loin d'être le seul.

Théorème 1. *(Indécidabilité du problème de l'arrêt) Savoir si un programme termine est un*

[2] Penser à la différence entre compétence et performance.

problème indécidable.

Proof. On va procéder par l'absurde (cf. annexe A.2.1). 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 l'utilisation de types, qui rendent la preuve donnée caduque. Il est cependant toujours possible de prouver l'indécidabilité de l'arrêt pour des programmes typés, de façon plus tordue cependant.

Au-delà de ces deux problèmes particulièrement connus, il existe une véritable armée de problèmes indécidables, comme le montre spécifié par le théorème de Rice :

Théorème 2. *(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.*

Proof. 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.

Intuitivement, tous ces problèmes sont indécidables sur tout modèle Turing-complet car ces derniers sont trop puissants, ou expressifs[3]. 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. Ce cours va s'intéresser à la famille des automates, particulièrement adaptée à l'analyse linguistique.

[3] Comme le disent tous les oncles Ben du monde, un grand pouvoir (expressif) implique une grande indécidabilité

Puisqu'on veut les analyser, on va d'abord définir la notion de langages - et donc de mots dans la section 2. On étudiera dans la section 3 un outil pour les décrire et manipuler, les expressions régulières. On pourra ensuite s'intéresser aux automates les plus simples dans la section 4, ainsi qu'à la notion connexe de grammaires formelles en 5. Enfin, on étudiera en 6 la façon dont ces différents outils s'imbriquent, et on *teasera* la suite du cours, en évoquant des extensions des automates.

Chapter 2

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.

2.1 Mots

Définition 2.1.1: 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 2.1.1. *Etant donné l'alphabet* $\Sigma = \{a, b, c\}$*, on peut construire une infinité de mots parmi lesquels*

- abc
- aab
- cc
- $abcabcacbacbacbabcabcabcabcabcabc$
- a

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

Définition 2.1.2: 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 2.1.3: Concaténation

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

Exemple 2.1.2. *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 2.1.4: Longueur d'un mot

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

Exemple 2.1.3. *Tout naturellement,*

- $|abc| = 3$
- $|abba| = 4$
- $|c| = 1$
- $|\epsilon| = 0$

Définition 2.1.5: 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.

Lemme 1. *Etant donnés deux mots w_1 et w_2, $|w_1.w_2| = |w_1| + |w_2|$.*

Proof. 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 2.1.6: 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 2.1.4. *On a*

- $|abc|_a = 1$
- $|abba|_b = 2$
- $|c|_a = 0$
- $|\epsilon|_a = 0$

Définition 2.1.7: Préfixe

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

Définition 2.1.8: Suffixe

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

Exemple 2.1.5. *Le mot abba admet comme préfixes ϵ, a, ab, abb et abba. Ses suffixes sont, quant à eux, ϵ, a, ba, bba et abba.*

Exercice 2.1.1. *Combien de préfixes et suffixes admet un mot w quelconque ?*

Définition 2.1.9: 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 2.1.6. *Les facteurs du mot abba sont ϵ, a, b, ab, ba, abb, bba et abba.*

Exercice 2.1.2. *Donner l'ensemble des facteurs du mot abbba.*

Exercice 2.1.3. *(*) 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 2.1.10: 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 2.1.7. *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 baaabaabbaa.*
- *ba n'est pas un sous-mot de aaabbb (l'ordre du sous-mot doit être préservé dans le mot)*

Exercice 2.1.4. *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 2.1.5. *Donner toutes les façons de voir abba comme sous-mot de baaabaabbaa (cf. exemple 2.1.7).*

Exercice 2.1.6. *Donner l'ensemble des sous-mots de abba*

Exercice 2.1.7. *(*) 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 2.1.8. *(*) Dans l'exercice 2.1.1, on demande le nombre exact de préfixes et suffixes d'un mot, alors que dans les exercices 2.1.3 et 2.1.7, on demande une borne, pourquoi ?*

Remarque A l'exception d'ϵ[1], tout mot admet au moins deux préfixes / suffixes / facteurs / sous-mots : ϵ et lui-même.

2.2 Langage

Un langage, c'est tout simplement un ensemble de mots. On distingue d'entrée deux langages un peu spéciaux, pour ne pas dire pathologiques.

Définition 2.2.1: Langage plein

Etant donné un alphabet Σ, on note $\mathbf{\Sigma}^*$ l'ensemble de tous les mots formés à partir de Σ.

Définition 2.2.2: Ensemble/Langage vide

On note $\emptyset$ l'ensemble vide, ou langage vide en ce qui nous concerne, 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.

[1] Il n'est bien sûr pas interdit de faire une analogie avec 1, seul entier à n'admettre qu'un diviseur.

Ces deux ensembles sont bien gentils, mais un peu radicaux. La question maintenant est donc de savoir comment on définit et parle de langages précis et plus "intermédiaires". En tout généralité, les ensembles peuvent être définis de façon **extensionnelle** ou **intentionnelle**.

Définition 2.2.3: 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 2.2.1. *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. Les définitions intentionnelles permettent elles de gérer ça, en utilisant des prédicats.

Définition 2.2.4: Prédicat

Un prédicat est une propriété, sur un ou plusieurs[a] éléments. On note les prédicats comme des fonctions, par exemple $P(x)$ un prédicat unaire et $Q(x, y)$ un prédicat binaire.

[a] Ou zéro, mais bon

Exemple 2.2.2. *On pose $P(x) \equiv x$ est de longueur paire. On a alors $P(ab)$ ou $P(\epsilon)$, mais pas $P(a)$ (ce qu'on note $\neg P(a)$).*

Exemple 2.2.3. *On pose $L(x, y) \equiv |x| = |y|$. On a alors $L(ab, ba)$, $L(aaaa, bbbb)$ et $\neg L(aba, ba)$.*

Exercice 2.2.1. *Donner l'ensemble des mots w tels que $L(w, \epsilon)$.*

Exercice 2.2.2. *Soit $\Sigma = \{a, b\}$, donner l'ensemble des mots w tels que $L(aba, w)$.*

Définition 2.2.5: 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 $Q(x)$ (par ex. 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 2.2.4. *On peut définir l'ensemble des mots de longueur paire $\{x \in \Sigma^* \mid P(x)\}$, ou plus simplement $\{x \mid P(x)\}$.*

Exercice 2.2.3. *Donner une définition de l'ensemble des mots de longueur 3 en utilisant le prédicat L défini dans l'exemple 2.2.3.*

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 "le mot w a une longueur 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.

Si plusieurs logiques équipées des bonnes primitives peuvent être utilisées, 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.

Chapter 3

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.

3.1 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 $+$.

3.1.1 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 3.1.1. *La regex a dénote le langage* $\{a\}$.

3.1.2 ., la concaténation

On peut heureusement concaténer des regex en utilisant à nouveau le symbole .[2]. 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 3.1.2. *La regex abca dénote l'ensemble* $\{abca\}$.

[1] On se trompera sans doute souvent en parlant d'"expression régulière"
[2] avec cependant un type légèrement différent, comme on le verra en 3.3

3.1.3 *, l'itération

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

Exemple 3.1.3. *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 3.1.4. *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 3.1.5. *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 3.1.6. *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 3.1.1. *Donner 5 autres mots appartenant au langage dénotée par l'expression de l'exemple 3.1.6.*

Exercice 3.1.2. *Pourquoi le changement de formulation dans les exemples 3.1.5 et 3.1.6 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 3.1.7. *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 3.1.7 et le mot *abbbabba*, on peut la voir comme

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

- $\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{bb}_{a^0b^2}}_{a^0b^2}\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{a}_{a^0}\underbrace{b}_{b^1}}_{a^0b^1}\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^0}}_{a^1b^0}}_{(a^*b^*)^5}}$
- $\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{b}_{a^0b^0}}_{a^0b^0}\underbrace{\underbrace{\underbrace{b}_{a^0}\underbrace{b}_{b^1}}_{a^0b^1}\underbrace{\underbrace{\underbrace{a}_{a^0}\underbrace{b}_{b^1}}_{a^0b^1}\underbrace{\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^1}}_{a^1b^1}\underbrace{\underbrace{\underbrace{b}_{a^0}\underbrace{a}_{b^1}}_{a^0b^1}\underbrace{\underbrace{a}_{a^1}\underbrace{b}_{b^0}}_{a^1b^0}}_{(a^*b^*)^7}}$

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.

3.1.4 +, la disjonction

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

Exemple 3.1.8. *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 3.1.9. *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 3.1.10. *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*

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

Exemple 3.1.11. *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 3.1.3. *Donner un mot acceptant deux dérivations avec la regex de l'exemple 3.1.10 (justifier en donnant les dérivations). Existe-t-il un autre mot admettant plusieurs dérivations ?*

Exercice 3.1.4. *Existe-t-il un mot acceptant plusieurs dérivations pour la regex de l'exemple 3.1.11 ?*

Exercice 3.1.5. *Donner un mot accepté par la regex de l'exemple 3.1.11 mais pas celle de l'exemple 3.1.10. Est-il possible de trouver un mot qui, à l'inverse, est accepté par la deuxième mais pas la première ?*

Exercice 3.1.6. *(*) Exprimer, en langue naturelle et de façon concise, le langage dénoté par la regex de l'exemple 3.1.7. 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 3.1.10, 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 3.1.11 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.

3.2 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 3.1: Syntaxe des expression régulières

La figure 3.1 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

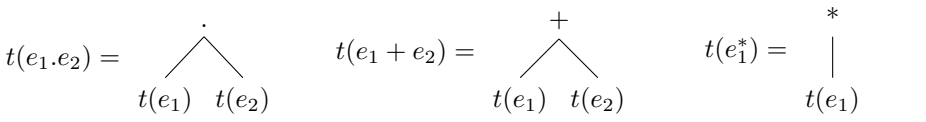
soit une expression rationnelle e_1, suivie de $+$, puis d'une expression rationnelle e_2

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 3.1 :

- $t(\epsilon)$ renvoie une feuille annotée par ϵ.
- $t(a)$ renvoie une feuille annotée par a.



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

Figure 3.2: Analyse syntaxique récursive de *regex*

- $t(e_1.e_2)$ renvoie un noeud . dont les descendants sont les arbres de e_1 et e_2, comme sur la figure 3.2a
- $t(e_1 + e_2)$ renvoie un noeud + dont les descendants sont les arbres de e_1 et e_2, comme sur la figure 3.2b
- $t(e_1^*)$ renvoie un noeud $*$ avec un seul descendant, l'arbre de e_1, comme sur la figure 3.2c

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 donc devoir définir les priorités entre les différentes opérations :

$$+ < . < *$$

Figure 3.3: Priorités pour les opérateurs d'expressions rationnelles

Concrètement, $+ <$. veut dire qu'une expression comme $a.b + c$ doit être interprétée comme $(a.b) + c$[3]. 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 3.2.1. *L'expression rationnelle $(aa)^* + (bb)^*$ peut être parsée comme*

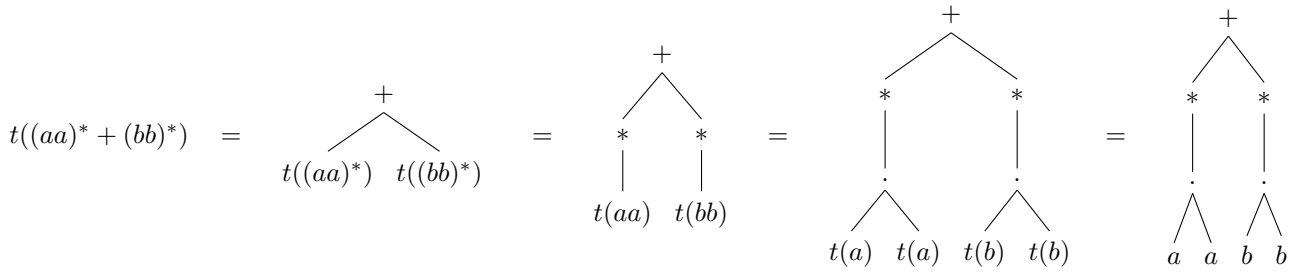
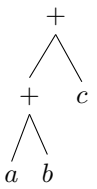


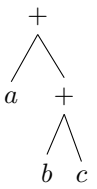
Figure 3.4: Analyse syntaxique de $(aa)^* + (bb)^*$

On remarquera bien sûr l'absence habile dans l'exemple 3.2.1 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 3.5 la première regex correspond.

[3] De la même façon que $a \times b + c$ se comprend comme $(a \times b) + c$



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



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

Figure 3.5: Ambiguïté syntaxique de $a + b + c$

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[4], 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 3.6

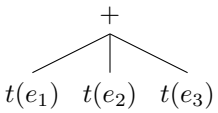
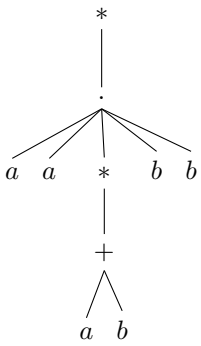


Figure 3.6: Ambiguïté syntaxique assumée de $a + b + c$

Exemple 3.2.2. *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.

3.3 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,

[4]Notez qu'en arithmétique, + et × sont également associatives

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 :

3.3.1 Les cas de base

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

3.3.2 Sémantique de la concaténation

Formellement, on a

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

Concrètement, ça veut dire que la sémantique de la concaténation de deux regex ($[\![e_1.e_2]\!]$) est le produit cartésien ($\times$) des sémantiques de e_1 ($[\![e_1]\!]$) et e_2 ($[\![e_2]\!]$), 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 code python suivant :

Exemple 3.3.1. *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 rigolos, on va avoir besoin d'ajouter des constructeurs à la sémantique.

3.3.3 Sémantique de la disjonction

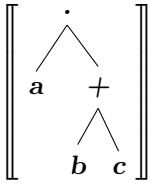
Formellement, on a

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

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

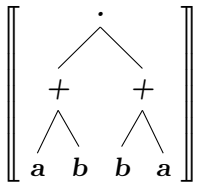
Remarque A partir d'ici et pour des raisons de mise en page, on ne mettra pas forcément tout sous forme d'arbres dans les exemples, et on comptera sur la capacité du lecteur ou de la lectrice à *parser* automatiquement toute expression rationnelle qu'il ou elle lit. Ne vous y trompez pas cependant :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

Exemple 3.3.2. *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 3.3.3. *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.

3.3.4 Sémantique de l'itération

Formellement, on a

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

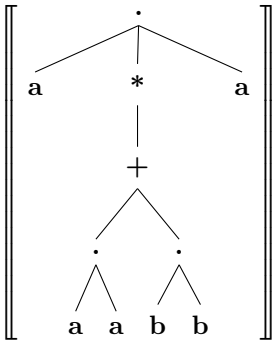
Concrètement, on fait l'union de $[\![e]\!]^n$ pour tous les entiers n, $[\![e]\!]^n$ étant n mots de $[\![e]\!]$ concaténés[5].

Exemple 3.3.4. *La sémantique de l'expression $a(aa+bb)^*a$ est*

3.4 Mise en application

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

[5]Les puissances sur les ensembles ont le même sens que sur les nombres, avec la multiplication remplacée par la concaténation



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

3.4.1 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 3.4.1. *Donner une regex pour les mots qui commencent par a.*

Exercice 3.4.2. *Donner une regex pour les mots qui finissent par b.*

Exercice 3.4.3. *Donner une regex pour les mots qui commencent par a finissent par b.*

Exercice 3.4.4. *Donner une regex pour les mots de longueur paire.*

Exercice 3.4.5. *Donner une regex pour les mots de longueur impaire qui contiennent au moins 4 lettres.*

Exercice 3.4.6. *Donner une regex pour les mots de longueur impaire, qui contiennent au moins 4 lettres, comment par a et finissent par b.*

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

Proof. 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* est donc étendue en pratique. 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 **sucre syntaxique**.

Chapter 4

Automates

Chapter 5

Grammaires formelles

Chapter 6

Théorème de Kleene et hiérarchie de Chomsky

Appendix A

Rappels mathématiques

A.1 Lexique

On rappelle quelques notions et notations de base.

A.2 Logique

A.2.1 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 $\perp$, ce qui veut dire soit que la logique est incohérente, soit que $\neg P$ est fausse, et donc que P est vraie.

Exemple A.2.1. *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.2.2. *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.3 Ensembles

lol

A.4 Prédicats

lol bis