Université Bordeaux Montaigne **Lionel Clément**
Formalisation Linguistique 1
Master RELAI–2015-2016 Département Langues et Civilisations

1 Système formel

Un système formel est un objet mathématique à deux faces. L'une est purement syntaxique. Elle dit comment les écritures du système formel sont possibles. L'autre est sémantique ; elle dit le sens de ces écritures. Les exemples sont nombreux : logique mathématique, théorie axiomatique des ensembles, etc.

L'intérêt d'un système formel est de permettre un raisonnement théorique sur l'écriture et non sur le sens de ces écritures. On peut par exemple automatiser le raisonnement et montrer les contradictions d'une théorie de façon reproductible et falsifiable sans utiliser l'intuition, les croyances ou la réthorique comme base argumentaire.

Un système formel est composé de
- Un alphabet fini de symboles
- Un procédé de construction des formules
- Un ensemble de formules axiomatiques (supposées correctes sans démontration)
- Un ensemble de règles de déduction (permettant de produire des formules correctes à partir d'autres formules correctes)

Par exemple la logique propositionnelle est un système formel qui contient :
- Un ensemble de lettres (opérateurs, parenthèses, constantes, variables, etc.)
- Un procédé de construction des formules :
  - Une constante ou une variable est une formule
  - Si f est une formule, alors $\neg(f)$ est une formule
  - Si f_1 et f_2 sont des formules, alors $(f_1 \to f_2)$, $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$ sont des formules
- Un ensemble d'axiomes :
  - $\vdash x \to (y \to x)$
  - $\vdash x \to (y \to z) \to ((x \to y) \to (x \to z))$
  - $\vdash (\neg y \to \neg x) \to (x \to y)$
- La règle de déduction *modus ponens*
  - $\frac{\vdash x, (\vdash x \to y)}{\vdash y}$

Le sens donné aux formules permet de motiver les axiomes et la règle de déduction. Les formules ont une interprétation notée ϕ dans $\{0, 1\}$ telle que

— $\phi(\neg(x))$ vaut $1 - \phi(x)$
— $\phi(x \rightarrow y)$ vaut 0 si et seulement si $\phi(x) = 1$ et $\phi(x) = 0$

2 Langage formel

La notion de **langage formel** nous permettra de modéliser des propriétés linguistiques tels que la structure syntaxique des phrases, la morphologie, les liens de dépendance entre les mots et les phrases, etc. Nous ne confondrons pas les langages logiques, mathématiques, de programmation informatique qui sont des **langages formels** et le **langage**.

Le langage formel est central et historique en logique et en informatique. Il l'est aussi dans la technique de programmation des ordinateurs qui utilise des *langages de programmation* depuis les années 1950. Il est donc naturel de trouver dans la littérature mathématique et informatique les notions d'analyse syntaxique qui servent en partie, mais pas seulement, le TAL. Nous trouverons par exemple des résultats théoriques qui intéressent le TAL dans le traité de théorie des langages « The theory of Parsing Translation and Compiling » de Alfred V. Aho et Jeffrey D. Ullman 1972, Prentice-Hall, Inc.

Par **langage formel**, nous entendons des ensembles de chaines de caractères. Une chaine de caractères appartient à un langage s'il a quelque propriété définitoire de ce langage. Cette propriété est précisément ce qu'on appelle une **grammaire**.

Un point de vue proche peut être suivi à propos des langues : le langage peut se modéliser par le fait que toutes les productions licites d'un locuteur sont grammaticales. La capacité de langage se résume alors à la capacité à produire les seules phrases grammaticales. Le programme de recherche linguistique appelé le *générativisme* propose de décrire une langue comme l'ensemble des productions grammaticales et de les distinguer de celles qui ne le sont pas. Il se propose de dire qu'une **grammaire** est le moyen de produire l'ensemble des séquences phoniques grammaticales et elles seules. On parle alors de la *compétence linguistique* du locuteur à produire de telles séquences. Les grammaires formelles que nous voyons dans ce chapitre permettent de modéliser cette compétence.

Les objets que l'on tente de modéliser du langage sont des éléments qui ne se définissent pas en propre, mais en différences. Ce sont les phonèmes, les monèmes, les graphèmes, etc. qui n'ont d'existence linguistique que par le fait qu'ils s'organisent pas oppositions et complémentarités.

L'objet de la formalisation qui nous intéresse ici est un flux, ou une séquence d'éléments qui se distinguent les uns des autres en nombre fini. Nous avons donc besoin d'une algèbre élémentaire pour construire les séquences.

Pour définir un langage, on se donnera deux outils : le premier est un ensemble fini d'éléments qu'on appellera un alphabet, le second est le moyen de mettre bout-à-bout ces éléments entre eux pour produire des séquences, la concaténation.

2.1 Monoïde libre

Un langage formel est donc un ensemble de séquences de lettres. Cet ensemble n'est pas nécessairement fini, mais les lettres sont données en nombre fini, ainsi que la taille des séquences.

L'ensemble des lettres est appelé **alphabet**. Il est noté $\mathbf{\Sigma}$. L'ensemble de toutes les séquences de lettres est noté $\mathbf{\Sigma}^*$.

Les chaînes de lettres se combinent entre elles grâce à la concaténation, une relation qui associe un élément unique de Σ^* à tout couple de $\Sigma^* \times \Sigma^*$ donné. Nous notons par la multiplication cette loi de composition. Elle permet de définir une séquence de lettres à partir de deux autres mises bout-à-bout. Séquenciellement elle définit toutes les suites possibles de séquences de lettres.

La concaténation respecte deux propriétés :

1. Elle est associative

$$\forall a, b, c \in \Sigma^*, (ab)c = a(bc)$$

2. Il existe un élément neutre dans Σ pour cette loi noté ϵ. Il s'agit de la séquence vide. Il est à la concaténation ce qu'est le un à la multiplication : un élément sans effet quand il est ajouté à une séquence existante.

$$\forall a \in \Sigma^*, a\epsilon = \epsilon a = a$$

Remarquons que la concaténation n'est pas commutative, et que les séquences de lettres n'ont pas d'inverse contrairement à ce que l'on trouve avec les nombres et la multiplication.

Σ^* permet de construire toutes les séquences de lettres Σ. Une telle structure algébrique s'appelle un **monoïde libre**.

La construction par induction du monoïde libre $\mathbf{\Sigma}^*$ se définit ainsi :

1. $\epsilon \in \mathbf{\Sigma}^*$
2. $\Sigma \subset \mathbf{\Sigma}^*$
3. Si $x, y \in \mathbf{\Sigma}^*$, alors $xy \in \mathbf{\Sigma}^*$

2.2 Langage formel sur Σ

Toute partie d'un monoïde Σ^* est un **langage formel** sur Σ.

On peut donc définir un langage en listant un ensemble de chaînes de lettres. Le point de vue linguistique générativiste équivalent consiste à dire que l'ensemble des productions grammaticales d'une langue donnée est cet ensemble. Il est cependant plus intéressant de le déterminer par les propriétés qui lui sont propres, c'est-à-dire en définissant une **grammaire**.

La structure des phrases, par exemple, ce qui met en rapport les chaînes de lettres entre elles par un ensemble de lois, feront partie des ces propriétés remarquables.

Un langage sur un alphabet Σ s'écrit

$$L_v = \{x \in \Sigma^* : P(x)\}$$

où $P(x)$ désigne les propriétés auxquelles doivent satisfaire les chaînes de lettres. La spécification de ces propriétés est ce qu'on appelle une **grammaire formelle**.

Il a été proposé par Noam Chomsky en 1957 (in Structures syntaxiques) de modéliser cette propriété par un mécanisme fini. Par ailleurs, nous rappelons qu'un objet infini n'a guère de chance d'être représenté *in extenso* par un mécanisme fini. La mécanisation d'un langage infini n'a donc pas de chance d'être réalisée autrement que par **induction**.

3 Langage rationnel et expressions régulières

Une expression régulière est une écriture qui permet de décrire immédiatement un motif, ou filtre pour un ensemble de chaines de caractères. L'objectif pratique est d'appliquer un traitement automatisé sur des textes ou toute autre séquence pour extraire un ensemble de chaines correspondant au motif.

Plus formellement, une expression régulière est un système formel qui dénote un langage (dit rationnel par définition).

La syntaxe des expressions régulières est la suivante :

Soit Σ un alphabet,

- La lettre vide ϵ est une expression régulière
- Tout élément de Σ est une expression régulière
- Si X et Y sont des expressions régulières, alors (XY) est une expression régulière
- Si X et Y sont des expressions régulières, alors $(X + Y)$ est une expression régulière

— Si X est une expression régulière, alors X^* est une expression régulière

La sémantique de cette notation est le langage qu'elle décrit, nous notons $L_\Sigma(X)$ pour désigner le langage décrit par l'expression régulière X sur l'alphabet Σ :

— $L_\Sigma(\epsilon) = \{\epsilon\}$
— $L_\Sigma(X + Y) = L_\Sigma(X) \cup L_\Sigma(Y)$
— $L_\Sigma(XY) = \{xy \in \Sigma^* : x \in L_\Sigma(X) \text{ et } y \in L_\Sigma(Y)\}$
— $L_\Sigma(X^*) = \cup_{i\in\mathbb{N}} L_\Sigma(X^i)$
$X^i = \underbrace{XX\dots X}_{i}$
$X^0 = \epsilon$

3.1 Langage engendré par une expression régulière

Une expression régulière dénote un langage. Ce langage est immédiatement définit par la sémantique du système formel.

Par exemple, l'expression régulière $a + b^*c$ dénote le langage $\{ac, abc, abbc, abbbc, abbb\dots bc, \dots\}$. Les mots préfixés par a, suffixés par c et qui contiennent zéro ou plusieurs b.

3.2 Extension des expressions régulières

Pour simplifier les notation, certains ajouts sont proposés aux expressions régulières. Ils n'étendent pas le système formel.

— $X^+ = XX^*$
— $X^? = \epsilon + X$
— $X^k = \underbrace{XX\dots X}_{k}$
— $X^{\{i,j\}} = \underbrace{XX\dots X}_{i} + \underbrace{XX\dots X}_{i+1} + \dots + \underbrace{XX\dots X}_{j}$
— $[a_1a_2\dots a_k] = a_1 + a_2 + \dots + a_k$ avec $a_i \in \Sigma$

4 Automates à nombre fini d'états

Les automates sont des objets mathématiques qui permettent de désigner des processus discrets et finis. Les processus passent d'un état à un autre étant donné un stimulus distingué d'autres stimuli donnés également en nombre fini.

La machine que représente l'automate possède un nombre fini d'états distingués les uns des autres et un ensemble fini de stimuli qui permettent de passer d'un état à un autre sans continuité. Ni les états, ni les stimuli ne

méritent une définition en propre. Un état de la machine est distingué de tous les autres états, un stimulus de tous les autres stimulus.

Les stimuli seront représentés par un alphabet fini Σ, les états par un ensemble fini Q.

Il reste à définir une relation de transition d'état à état selon les éléments de Σ et à dire les conditions de départ et d'arrêt de la machine pour être complet.

4.1 Définition formelle

Un automate à nombre fini d'états est un quintuplet $A = (Q, \Sigma, q_0, F, \phi)$ où

Q est un ensemble fini d'états

Σ est un alphabet fini

q_0 est un élément distingué de Q, appelé état initial

F est une partie de Q, dont les éléments sont appelés états terminaux

ϕ est une relation, qui à tout couple (q, a) de $Q \times \Sigma$, associe un élement de Q

L'automate est dit **déterministe** si ϕ est une fonction, qui à tout couple (q, a) de $Q \times \Sigma$, associe un élément unique de Q.

L'automate est dit **ϵ-automate** si ϕ est une relation, qui à tout couple (q, a) de $Q \times (\Sigma \cup \{\epsilon\})$, associe un élément ou plusieurs éléments différents de Q

4.2 Automate reconnaissant un langage

Une séquence $a_0 a_1 a_2 \dots a_k$ de Σ^* est reconnue par l'automate si et seulement s'il existe une suite d'états $(q_0, q_1, q_2, \dots, q_k)$ où

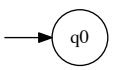
q_0 est l'état initial

q_k appartient aux états terminaux

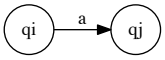
$\forall i \in [0, k-1]$, $\phi(q_i, a_i) = q_{i+1}$

Un automate à nombre fini d'états sur un alphabet Σ permet ainsi de décrire un ensemble de chaînes de Σ^*. Voici donc une seconde façon de décrire un langage.

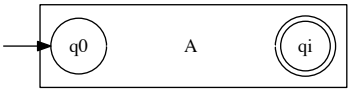
On représente graphiquement un automate comme ceci

l'état initial q_0 

Un état final q_i 

Une transition $\phi(q_i, a) = q_j$ 

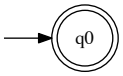
Un automate `A` avec q_0 comme état initial et q_i comme état final

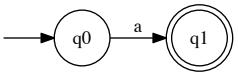


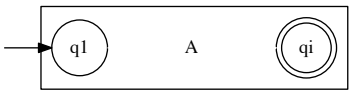
4.3 Automate reconnaissant un langage régulier

A toute expression régulière, il existe un automate équivalent, c'est-à-dire qui reconnait le même langage.

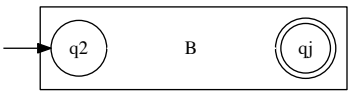
Procédons par construction inductive :

1. l'automate qui reconnaît $L_\Sigma(\epsilon)$ est 

2. l'automate qui reconnaît $L_\Sigma(a)$ est 

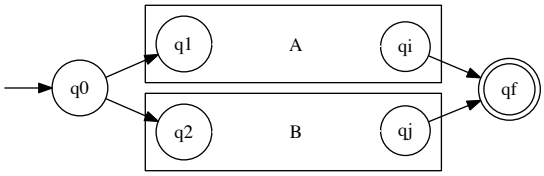
3. Soit 

l'automate qui reconnaît $L_\Sigma(A)$

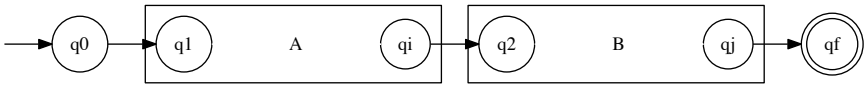
et 

l'automate qui reconnaît $L_\Sigma(B)$

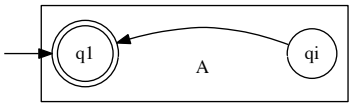
L'automate qui reconnaît $L_\Sigma(A + B)$ est



4. L'automate qui reconnaît $L_\Sigma(AB)$ est



5. L'automate qui reconnaît $L_\Sigma(A^*)$ est



4.4 Suppression des transitions vides

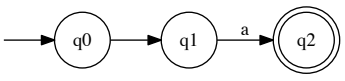
Pour tout automate contenant des transitions vides, il existe un automate équivalent (qui engendre le même langage).

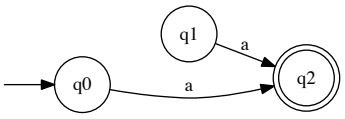
Voici comment le construire :

Soit un automate $A = (Q, \Sigma, q_0, F, \phi)$ contenant des transitions vides, c'est-à-dire des transitions $\phi(q_i, \epsilon) = q_j$. On contruit l'automate $A' = (Q, \Sigma, q_0, F, \phi')$ où $\phi'(q_i, \alpha) = q_k$ si

1. $\phi(q_i, \alpha) = q_k$
2. Si $\phi(q_i, \epsilon) = q_{j_1}, \phi(q_{j_1}, \epsilon) = q_{j_2}, \ldots \phi(q_{j_k}, \epsilon) = q_j$, et $\phi(q_j, \alpha) = q_k$

Illustration graphique :





4.5 Déterminisation d'un automate

Tout automate fini sans transition vide A est équivalent à un automate déterministe fini A'. Le principe pour construire A' consiste à lui attribuer comme états possibles, l'ensemble de tous les sous-ensembles des états de A.

Les transitions sont celles qui permettent de passer d'un sous ensemble à un autre dans l'automate A.

Soit $A = (Q, \Sigma, q_0, F, \phi)$ un automate sans transition vide.

$A' = (Q', \Sigma, q'_0, F', \phi')$ se construit ainsi :

— $Q' = \mathcal{P}(Q)$

— $q'_0 = \{q_0\}$

— $\forall q_i \in F, q_i \in q'_i \Rightarrow q'_i \in F'$ (un état q'_i de A' est final s'il contient un état final de A)

— $\phi'(q'_i, \alpha) = q'_j \Leftrightarrow \forall q_i \in q'_i, \forall q_j \in q'_j, \phi(q_i, \alpha) = q_j$

Exemple :

Soit l'automate A, l'automate A' reconnaît le même langage.

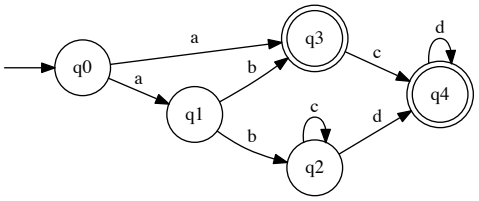


FIGURE 1 – Automate A

5 Systèmes de réécriture

5.1 Grammaire formelle : grammaire de réécriture

Une grammaire formelle est définie par le quadruplet (Σ, N, R, S) où :

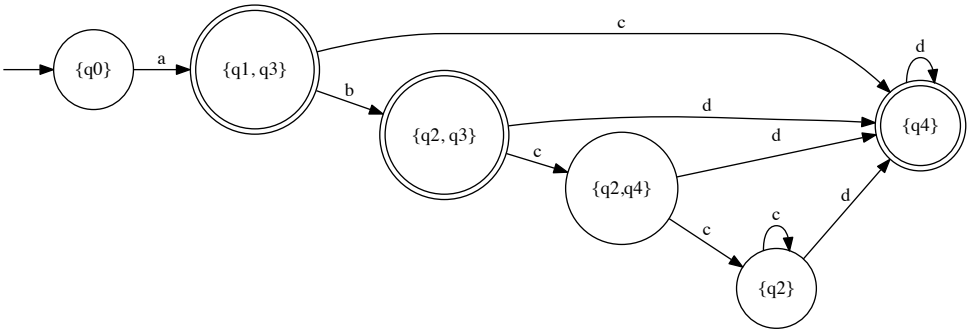


FIGURE 2 – Automate A'

1. Σ est un alphabet fini (on parle de symboles terminaux)
2. N est un ensemble fini de termes distincts de Σ (on parle de symboles non terminaux)
3. S est un élément distingué de N ; le terme initial de la grammaire
4. R est une relation de **réécriture** $\{(\alpha, \beta)\}$, où $\alpha \in (\Sigma \cup N)^* - \{\epsilon\}$, et $\beta \in (\Sigma \cup N)^*$. On écrit $\alpha \to \beta$.

Une grammaire formelle permet de définir les éléments du langage par une opération qui fait intervenir une succession de réécritures depuis le terme initial jusqu'aux chaînes de lettres.

Dérivation On définit une relation $\Rightarrow$ entre deux éléments du monoïde libre $(\Sigma \cup N)^*$

$$\forall \mu_1, \alpha, \mu_2, \beta \in (\Sigma \cup N)^*, \mu_1 \alpha \mu_2 \Rightarrow \mu_1 \beta \mu_2$$

vérifiée si et seulement s'il existe une règle $\alpha \to \beta$ de R

On définit une **dérivation** par la fermeture réflexive-transitive $\overset{*}{\Rightarrow}$

La fermeture réflexive-transitive d'une relation R, noté R^*, se définit ainsi : $\forall x, y \in E, xR^*y$ si et seulement si

1. $\forall x \in E, xR^*x$
2. $\forall x, y, z \in E, xRy$ et $yRz \Rightarrow xR^*z$

Autrement dit, tout élément se dérive vers lui-même (réflexivité), et un élément x se dérive vers un élément z s'il se dérive vers un élément y qui se dérive vers z (transitivité).

Langage engendré par une grammaire Le langage engendré par la grammaire G, noté $\mathcal{L}_G$, est l'ensemble de toutes les chaînes de lettres qui dérivent de l'élément initial.

On note :

$$\mathcal{L}_G = \{\alpha \in \Sigma^* : S \overset{*}{\Rightarrow} \alpha\}$$

Types de grammaires

1. Les grammaires formelles les plus simples qu'on puisse imaginer sont les grammaires rationnelles. Il s'agit de grammaires telles que les règles sont de la forme

   $$A \to a$$

   $$A \to aB$$

   (resp. $A \to a$ ou $A \to Ba$ pour les grammaires rationnelles *à droite*).
   Avec $A, B \in N$ et $a \in \Sigma$.

   Les grammaires rationnelles permettent de décrire les langages rationnels tout comme les expressions régulières le font. Il est donc toujours possible de construire un automate déterministe à nombre fini d'états qui génère le même langage qu'une grammaire rationnelle.
2. Les grammaires qui n'ont qu'un élément de N en partie gauche sont dites *indépendantes du contexte*. La dérivation $\mu_1\alpha\mu_2 \Rightarrow \mu_1\beta\mu_2$ est vérifiée si et seulement si la règle $\alpha \to \beta$ appartient à R ; ceci indépendamment du contexte de α, c'est-à-dire indépendamment de μ_1 ou de μ_2.
3. Les grammaires contextuelles sont celles qui acceptent les règles

   $$x\alpha y \to x\beta y$$

   et pour lesquelles la séquence α n'est pas plus longue que la séquence β. Ici en revanche, une dérivation $\mu_1\alpha\mu_2 \Rightarrow \mu_1\beta\mu_2$ est vérifiée étant connus μ_1 ou μ_2.
4. Les autres grammaires sont dites *non contraintes* ou *récursivement énumérables*. Leur seule contrainte est d'avoir une séquence non vide en partie gauche. Elles ne nous intéressent pas ici.

6 Analyse syntaxique automatique

Nous distinguerons les reconnaisseurs automatiques des analyseurs syntaxiques automatiques. Les premiers disent si une séquence de lettres est un élément du langage ou non, les seconds, en plus de cette réponse, y associent une ou plusieurs dérivations.

La reconnaissance des langages a une implication théorique, mais nous intéressera moins pour les applications du Traitement Automatique des Langues.

On peut distinguer plusieurs méthodes pour construire des analyseurs syntaxiques.

— Ascendant – descendant

Un analyseur descendant utilise l'hypothèse que le terme initial sera dérivé vers la séquence de lettres donnée en input à l'analyseur. Le mécanisme tente de remplacer le terme initial S par la séquence de termes α de la partie droite de la règle $S \rightarrow \alpha$. Puis récursivement, il fait l'hypothèse que les proto-phrases [1] induites par cette hypothèse seront elles-mêmes dérivées.

Un analyseur ascendant, au contraire, part de la séquence de lettres jusqu'au terme initial en cherchant à trouver les parties droites des règles dans la séquence de la proto-phrase courante pour les remplacer par les parties gauches.

— Déterministes – non déterministes

Que l'analyseur soit descendant ou ascendant, le choix de la règle à appliquer est parfois problématique. A un instant donné de l'analyse, il se peut qu'une proto-phrase puisse correspondre à plusieurs dérivations qui ne vont pas toutes aboutir au résultat [2]. Un analyseur déterministe applique seulement les dérivations qui conduisent à des résultats. Deux alternatives permettent de construire des analyseurs non déterministes. La première est peu heureuse : elle consiste à faire machine arrière en cas d'échec pour explorer les autres dérivations laissées en suspend quand plusieurs choix étaient possibles. Une autre plus efficace, qu'on verra avec deux algorithmes (Earley et CKY), applique toutes les dérivations en parallèle au même instant. Certaines vont aboutir, d'autres non. On peut dire que cette dernière méthode est gourmande en place mais satisfaisante en durée du processus.

1. Une proto-phrase est une séquence de terminaux et de non terminaux qui dérive du terme initial. Soit $x \in (\Sigma \cup N)^* / S \stackrel{*}{\Rightarrow} x$

2. Si plusieurs dérivations gauches (resp. droites) peuvent aboutir à des résultats, la grammaire est dite ambiguë.