LANGAGES - GRAMMAIRES - AUTOMATES

Marie-Paule Muller

Version du 14 juillet 2005

Ce cours présente et met en oeuvre quelques méthodes mathématiques pour l'informatique théorique.

Ces notions de base pourront servir d'entrée en matière avant d'aborder un cours de compilation. Elles sont introduites ici sans aucun prérequis, afin d'être accessibles à tout lecteur débutant sur ce sujet.

Table des matières.

INTRODUCTION

Un *compilateur* est un utilitaire de traduction permettant, à partir d'un programme écrit dans un langage de "haut niveau" ou, du moins, compréhensible par un programmeur humain (C/C++, Pascal, Algol, FORTRAN, assembleur, ...), de vérifier la syntaxe du programme de départ, puis de produire un fichier en un langage de niveau plus bas, par exemple un fichier en code objet, exécutable par le système d'exploitation.

Le premier langage de haut niveau qui a été écrit est le FORTRAN (« Mathematical FORmula TRANslating System ») ; son compilateur a été conçu et écrit dans les années 1954-57 par une équipe de pionniers super-programmeurs conduite par John W. Backus. Son écriture, soit 25 000 lignes de code machine enregistrées sur bande magnétique, a nécessité un travail équivalent à 18 hommes-années, bien plus important que ce que le projet initial prévoyait ; mais la direction d'IBM, dont dépendait ce projet, a eu l'intelligence de laisser le groupe libre de poursuivre son effort comme il l'entendait. Coup d'essai, coup de maître : le langage FORTRAN I a eu un succès énorme, et son compilateur a gardé durant vingt ans le record d'optimisation du code objet produit.

Plus tard, le compilateur du PASCAL a été écrit en *auto-amorçage* : conçu « à la main » pour 60% environ du langage, le reste a été produit par ce qui était déjà compilé. De nombreux autres compilateurs ont suivi (par exemple YACC, "Yet Another Compiler of Compiler").

Durant les mêmes années 1955-65, des linguistes, philosophes et mathématiciens ont défriché la partie théorique en proposant une description et une classification des langages et des grammaires, pour les diverses langues naturellement utilisées puis pour les langages de programmation. Parmi eux, citons le linguiste Noam Chomsky, le mathématicien logicien Stephen Kleene, l'informaticienne Sheila Greibach, dont nous reverrons les noms dans ce cours.

1. Les tâches d'analyse d'un compilateur.

Les premières tâches d'un compilateur sont de faire :

- *l'analyse lexicale* : reconnaître les éléments constitutifs de la chaîne entrée, c'est-à-dire du code source, et en dresser la liste.
- *l'analyse syntaxique* : vérifier la conformité avec les règles de constitution du code. Par exemple, l'expression **(A+B)) = C** est syntaxiquement incorrecte (parenthèses…).
- *l'analyse sémantique* : analyser le sens et fixer une interprétation. Par exemple dans « `si A alors si B alors C sinon D` », choisir à quel `si` se rapporte le `sinon`.

Dans ce qui suit, nous nous occuperons essentiellement de l'analyse syntaxique.

2. La notion de grammaire et d'analyse syntaxique.

Considérons, par exemple, la phrase suivante :

LE VIEUX CHAT ATTRAPE LE PETIT RAT

Le but est de construire une grammaire qui permette de produire cette phrase.

Il faut tout d'abord préciser les « ingrédients » nécessaires : ce sera l'*alphabet des symboles* ou *lexique*. Dans notre cas, nous pouvons prendre comme alphabet l'ensemble

$$\Sigma = \{\text{LE, VIEUX, PETIT, CHAT, RAT, ATTRAPE}\}$$

(il contient ici six symboles).

Noter que le terme « alphabet » n'a pas ici le sens habituel A, B, C,…. : en fait, l'alphabet contient les « briques de base » avec lesquelles on peut former… tout ce qu'on veut pouvoir former ! Dans un langage de programmation par exemple, les mots réservés, balises, etc… comme `program`, `real`, `<head>`, `</head>`, … sont dans l'alphabet de symboles.

Voyons maintenant comment ces symboles sont assemblés. Dans la structure de la phrase, on peut distinguer :

- un groupe sujet
- un verbe
- un groupe complément d'objet (CO)

Les groupes sujet et CO sont eux-mêmes des groupes nominaux : un groupe nominal est formé d'un article suivi d'un nom, lui-même précédé ou suivi d'adjectifs.

Voici un exemple de (petite) grammaire pouvant produire notre phrase ; elle a onze *règles de grammaire* (on dit aussi *de production,* ou *de réécriture*) :

1 . <phrase> → <groupe sujet> <verbe> <groupe CO>
2. <groupe sujet> → <groupe nominal>
3. <groupe CO> → <groupe nominal>
4. <groupe nominal> → <article> <nom>
5. <groupe nominal> → <article> <adjectif> <nom>
6. <article> → LE
7. <nom> → CHAT
8 . <nom> → RAT
9. <adjectif> → VIEUX
10. <adjectif> → PETIT
11. <verbe> → ATTRAPE

- Le point de départ s'appelle *l'axiome.* Dans notre exemple, c'est <phrase>.

- Les *variables* sont les « ingrédients » qui peuvent encore être remplacés par d'autres (<phrase>, <article>, <groupe CO>, etc…).

- Les règles 1 à 5 sont des règles *syntaxiques* (la première règle concerne toujours l'axiome). Les règles 6 à 11 sont des règles *complètement terminales* (ou : *lexicales*).

- Le *langage engendré* par la grammaire est l'ensemble de toutes les phrases que l'on peut produire à partir de l'axiome en utilisant des règles de grammaire, une phrase étant une chaîne de symboles.

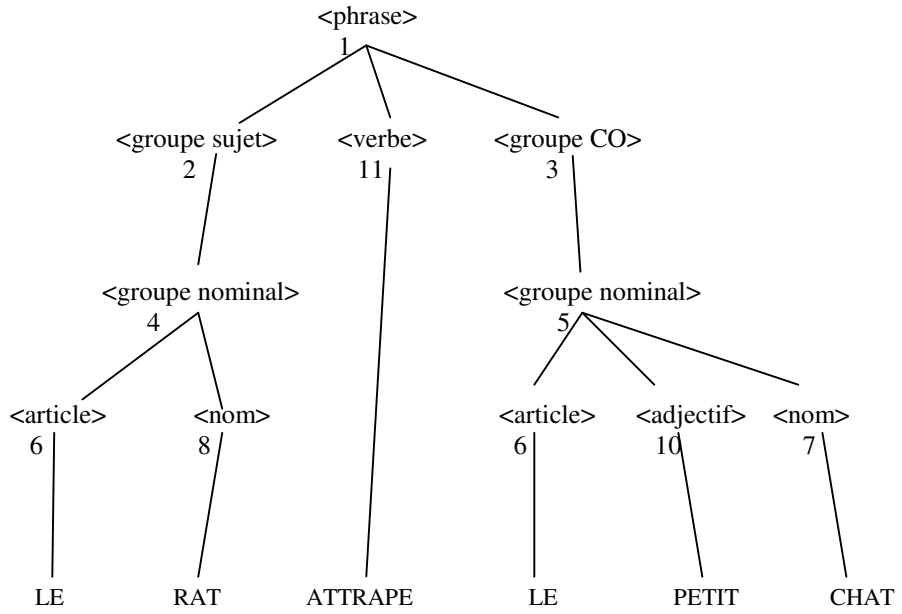
Voici maintenant quelques exemples de phrases « grammaticalement correctes », c'est-à-dire des chaînes de symboles que l'on peut produire à partir de l'axiome, en utilisant les règles précédentes :

LE RAT ATTRAPE LE PETIT CHAT
LE CHAT ATTRAPE LE VIEUX CHAT

On peut associer à chaque phrase ainsi produite un *arbre de dérivation* où figurent les variables auxquelles on a appliqué des règles de grammaire.

Nous avons ajouté aux nœuds de cet arbre de dérivation, pour faciliter la compréhension, les numéros des règles qui ont été appliquées aux variables.

Un arbre de dérivation pour la phrase LE RAT ATTRAPE LE PETIT CHAT est représenté dans la figure suivante.



LANGAGES – LANGAGES REGULIERS

1. Définitions.

L'*alphabet des symboles* est un ensemble fini. On le notera en général Σ, dans la suite du cours.

Exemple 1. Σ = { LE, CHAT, ..., VIEUX, ...} cf. l'introduction.
Exemple 2. Σ ={0, l, 2, ..., 9, +, *, -, / , (,) }. Cet alphabet permet d'écrire les expressions arithmétiques sur les nombres entiers, avec les quatre opérations et les parenthèses. Anticipons un peu : le premier problème sera de pouvoir distinguer si une expression est « correctement écrite » ; par exemple, **2*(31-6)+8** est correcte, mais **2(31-6)+8** ou encore **(21+)*4** ne le sont pas. Le deuxième problème sera, pour une expression correctement écrite, de la calculer selon les règles de l'art, c'est-à-dire de respecter les priorités (donner la priorité à la multiplication sur l'addition, calculer d'abord ce qui est entre parenthèses, ...).

Une *chaîne* est une suite finie de symboles.
La *longueur* d'une chaîne est le nombre de ses symboles.

Exemples. LE LE CHAT est une chaîne de longueur 3 sur l'alphabet de l'exemple 1.
Les expressions arithmétiques (correctes ou non !) sont des chaînes sur l'alphabet de l'exemple 2.

La *chaîne vide,* notée λ, ne contient aucun symbole ; sa longueur est nulle. On verra qu'elle est particulièrement utile !

Σ^n est l'ensemble de toutes les chaînes de longueur n.
$\Sigma^0 = \{ \lambda \}$, par convention. Attention, cet ensemble Σ^0 n'est pas vide : il contient la chaîne vide.
Σ^* est la réunion des Σ^n pour $n \geq 0$. C'est donc l'ensemble de toutes les chaînes, chaîne vide comprise.
Σ^+ est la réunion des Σ^n pour $n > 0$.

On peut *concaténer* des chaînes de symboles, c'est-à-dire les « coller » les unes derrière les autres, de la même façon que plusieurs textes peuvent être assemblés les uns derrière les autres pour former un nouveau texte. Si u et v sont deux chaînes de symboles, leur concaténation sera notée u.v ou plus simplement uv .

Un *langage* sur Σ est un sous-ensemble de Σ^*.

Exemples. Sur l'alphabet Σ = {a, b, c, d}, les ensembles suivants sont des langages :

- L_1 = {ac, abbc}
- l'ensemble L_2 de toutes les chaînes qui commencent par a.
- L'ensemble L_3 de toutes les chaînes de longueur paire.

Notation. Afin de faciliter la lecture, nous adopterons en général une notation a, b, c, … pour des symboles de Σ et u, v, w, x, … pour des chaînes de symboles.

2. Opérations sur les langages.

Un alphabet de symboles Σ étant fixé, voyons les opérations que nous appliquerons aux langages sur Σ .

- **Opérations ensemblistes usuelles (réunion, intersection, complémentaire).**

Comme les langages sur Σ ne sont rien d'autre que des sous-ensembles de Σ^*, les opérations habituelles que l'on connaît sur les sous-ensembles s'appliquent à ces langages.

- **Concaténation de langages**.

Comme nous savons concaténer des chaînes de symboles, nous pourrons également *concaténer deux langages*, c'est-à-dire collecter dans un nouvel ensemble, noté $L_1.L_2$, toutes les chaînes que l'on peut obtenir en concaténant une chaîne de L_1 avec une chaîne de L_2 .

Par récursivité, nous pourrons concaténer aussi un nombre fini de langages.

En particulier, si L est un langage, on notera

$$L^0 = \{ \lambda \}$$
$$L^1 = L$$
$$L^2 = L . L$$

puis on définit de manière récursive

$$L^n = L^{n-1}. L \quad \text{(concaténation de n exemplaires de } L \text{)}$$

- **Etoile de Kleene d'un langage.**

Pour un langage L donné, l'*étoile de Kleene* L^* est le langage obtenu en réunissant tous les langages L^n $(n \geq 0)$: $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$.
On note $L^+ = L.L^* = L^1 \cup L^2 \cup L^3 \ldots$.

Convention de priorité pour ces opérations.

On convient que l'étoile de Kleene est prioritaire sur la concaténation, qui est elle-même prioritaire sur les opérations ensemblistes ; pour modifier un ordre de priorité, on se sert de parenthèses. Formellement, ces règles sont donc analogues à celles que l'on a en arithmétique.

Exemple. Sur l'alphabet $\Sigma = \{0, 1, 2, ...,9\}$, le langage $\{1, 2, ..., 9\}.\Sigma^* \cup \{0\}$ est le langage de tous les nombres entiers naturels, écrits en base 10 et sans 0 inutiles à gauche.

3. Langages réguliers (Kleene, 1956).

Définition. Un langage L sur Σ est *régulier* si on peut l'obtenir par récursivité :

- à partir des seuls langages " de base " :
  - langage vide : ϕ
  - langage $\{\lambda\}$ (le langage ne contenant que la chaîne vide)
  - langages de la forme $\{a\}$, avec $a \in \Sigma$
- et à l'aide des seules opérations
  - réunion
  - concaténation
  - étoile de Kleene

« Par récursivité » veut dire qu'on applique un nombre fini de fois de telles opérations, à partir des langages " de base ".

Exemple. Sur $\Sigma = \{a, b, c\}$, le langage L des chaînes commençant par a est régulier, puisqu'on peut en donner une expression régulière en l'écrivant sous la forme

$$L = \{a\}.(\{a\} \cup \{b\} \cup \{c\})^*$$

Notation. Par abus de notation, on peut omettre les accolades pour alléger l'écriture. Le point de concaténation n'est pas toujours écrit non plus, si cette omission peut se faire sans ambiguïté :

$$L = a\ (a \cup b \cup c)^*$$

Remarque. Attention ! dans la définition, l'intersection ne figure pas comme une opération autorisée. En fait, on peut *démontrer* (mais c'est difficile…) que l'intersection de deux langages réguliers est encore un langage régulier :

Proposition 1.

1) Tout langage fini est régulier.
2) Si L est un langage régulier, alors L^* est régulier.
3) Si L_1 et L_2 sont des langages réguliers, alors les langages $L_1 \cup L_2$ et $L_1.L_2$ sont aussi réguliers.

Proposition 2.

Si L_1 et L_2 sont des langages réguliers, alors $L_1 \cap L_2$ est régulier.

La proposition 1 découle directement de la définition. Quant à la proposition 2, elle ne pourra être prouvée que bien plus tard, dans la suite du cours.

GRAMMAIRES ALGEBRIQUES (dites aussi : « hors contexte »)

Le problème : un langage L étant fixé, comment savoir si une chaîne w donnée appartient à L ?

Une solution possible : décrire une grammaire adéquate engendrant L , grammaire qui pourrait permettre

- de produire toutes les chaînes de L
- de tester une chaîne, en renvoyant un message d'erreur si elle n'est pas dans L.

1. Définition d'une grammaire algébrique.

Définition. Une *grammaire algébrique* se définit par la donnée de :

- Σ : un ensemble fini de *symboles terminaux*
- V : un ensemble fini de *variables*
- $S \in V$: une variable particulière, appelée *axiome* (« Start symbol »)
- P : un ensemble fini de *règles de production* (ou : de *réécriture*).
  Ces règles doivent être de la forme $A \to u$, où $A \in V$ et $u \in (\Sigma \cup V)^*$

En pratique, les variables seront répertoriées sous la forme d'une liste (elles seront donc ordonnées), mais la première de ces variables est toujours l'axiome. Les règles de production seront elles aussi ordonnées, en respectant l'ordre des variables auxquelles elles s'appliquent. La première règle concerne donc toujours l'axiome. L'ordre ainsi choisi aura une incidence sur le déroulement des algorithmes qui s'appliqueront aux grammaires.

Règle vide : règle de la forme $A \to \lambda$ (rappelons que λ est la chaîne vide).

De telles règles sont très utiles car elles autorisent certaines possibilités sans y obliger. Par exemple, un groupe nominal *peut* contenir des adjectifs, mais ce n'est pas une obligation ; un programme *peut* contenir des procédures, etc…

Règle directement récursive : règle de la forme $A \to vAw$ où $v, w \in (\Sigma \cup V)^*$

Autrement dit, la variable A figure encore dans la chaîne de réécriture de A .

De telles règles permettent de multiplier des occurrences de symboles, et de rallonger indéfiniment des expressions. En effet, une telle règle permet d'obtenir, à partir de A , les chaînes vAw , vvAww , vvvAwww , …

Les parenthèses dans les expressions arithmétiques (x) , ((x)) , (((x))),… en sont un exemple. De manière plus générale, si on remplace une variable d'une expression arithmétique par une (autre) expression arithmétique, on obtient une (nouvelle) expression arithmétique correctement écrite.

Notations.

Dans la suite du cours, afin de rendre le texte plus lisible, les variables d'une grammaire seront en général désignées par des lettres majuscules S, A, B, C, …

Pour les symboles terminaux, nous aurons les minuscules a, b, c, … sauf cas particuliers.

Les notations x, y, u, v, … désigneront souvent des chaînes de $(\Sigma \cup V)^*$**,** formées de symboles terminaux et/ou de variables.

Les « grammaires » seront toujours des grammaires algébriques.

Un exemple de grammaire. G : $S \to abSA$

$S \to \lambda$

$A \to Aa$

$A \to \lambda$

Ici, l'ensemble des variables est $V = \{S, A\}$, et l'axiome est S .
L'ensemble Σ des symboles terminaux est sous-entendu : $\Sigma = \{a, b\}$

La notation plus synthétique que nous adopterons consiste à se servir d'une barre verticale comme séparateur entre des règles pour une même variable :

G : $S \to abSA \mid \lambda$
$A \to Aa \mid \lambda$

Ces règles de grammaire seront supposées ordonnées (numérotées) : 1 | 2
3 | 4

2. Dérivations. Langage engendré.

Dans ce qui suit, A est une variable et x, y, u , $v \in (\Sigma \cup V)^*$.
En appliquant une règle $A \to u$ à la variable A qui figure dans une chaîne $v = xAy$, on produit la chaîne $w = xuy$. Cette opération est dite *dérivation*, et est notée $xAy \Rightarrow xuy$. On peut répéter ceci autant de fois que l'on veut en appliquant de nouveau une règle à la chaîne ainsi produite.

Définition. Une chaîne w *dérive* d'une chaîne v si, à partir de v , on peut arriver à w par un nombre fini d'applications de règles de grammaire :

$v \Rightarrow .... \Rightarrow w$, ce qu'on notera également $v \Rightarrow w$

On peut préciser le nombre d'applications de règles au-dessus de la double flèche, si cela s'avère utile (pour certaines preuves par récurrence, par exemple…) :

$v \overset{n}{\Longrightarrow} w$ (ici, on a fait n applications de règles de grammaire).

Définition. Le *langage engendré* par une grammaire G est l'ensemble de toutes les chaînes terminales (autrement dit, dans Σ^*) qui dérivent de l'axiome S . Il est noté L(G) .

Exemple 1. Soit G : $S \to abSA \mid \lambda$

$A \to Aa \mid \lambda$

Numérotons les règles de grammaire dans leur ordre de lecture.
Voici une suite de dérivations, qui prouve d'ailleurs que la chaîne ababa est dans L(G) :

$$\mathbf{S} \underset{1}{\Rightarrow} ab\mathbf{S}A \underset{4}{\Rightarrow} ab\mathbf{S} \underset{1}{\Rightarrow} abab\mathbf{S}A \underset{3}{\Rightarrow} abab\mathbf{S}Aa \underset{2}{\Rightarrow} abab\mathbf{A}a \underset{4}{\Rightarrow} ababa \in \Sigma^*$$

Pour chaque dérivation, on a mis en gras (l'occurrence de) la variable qui a été réécrite et on a indiqué sous la flèche le n° de la règle appliquée.

Exemple 2. Le langage engendré par la grammaire du PASCAL est l'ensemble de tous les programmes qui sont correctement écrits en code source PASCAL.

Remarque. Deux grammaires différentes peuvent engendrer le même langage.
Cette remarque est même décisive pour la suite ! En effet, on écrit une première grammaire compréhensible, « intuitive », engendrant le langage voulu, c'est-à-dire adaptée aux objectifs poursuivis (par exemple du calcul scientifique avec une précision paramétrable aussi grande que l'on veut ; de la gestion bancaire ; etc.). Puis il faut la transformer (à l'aide d'algorithmes ad hoc) afin d'obtenir une autre grammaire engendrant le même langage, mais qui aura l'avantage sur la première de permettre une compilation efficace et rapide.

3. Arbre de dérivation. Dérivations à gauche.

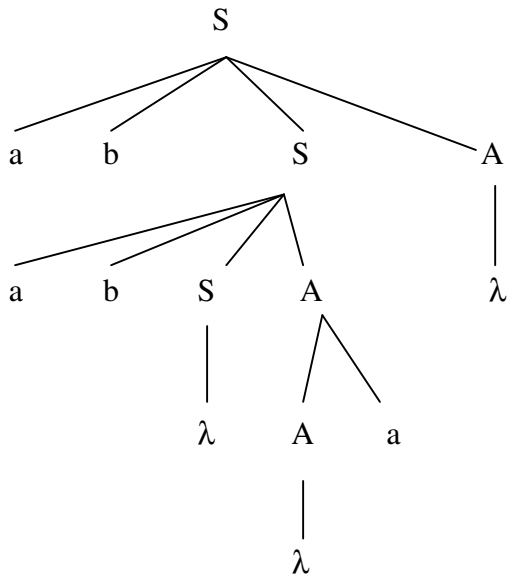
On peut tracer un a*rbre de dérivation* associé à toute suite de dérivations. Chaque nœud représente un exemplaire d'une variable réécrite, avec le numéro de la règle de grammaire appliquée.

Exemple. Reprenons la grammaire G : $S \to abSA \mid \lambda$
$A \to Aa \mid \lambda$

ainsi que la suite de dérivations

$$\mathbf{S} \underset{1}{\Rightarrow} ab\mathbf{S}\mathbf{A} \underset{4}{\Rightarrow} ab\mathbf{S} \underset{1}{\Rightarrow} abab\mathbf{S}\mathbf{A} \underset{3}{\Rightarrow} abab\mathbf{S}Aa \underset{2}{\Rightarrow} abab\mathbf{A}a \underset{4}{\Rightarrow} ababa$$

Voici l'arbre de dérivation qui lui est associé



En lisant les symboles qui figurent aux extrémités des branches, de gauche à droite, on obtient la chaîne ababa

Remarque 1. Des suites de dérivations différentes peuvent être associées à un même arbre de dérivation. En effet, il apparaît clairement sur cet exemple que l'ordre de certaines des dérivations peut être échangé sans que l'arbre en soit modifié.

Définition. Une *dérivation à gauche* est une dérivation qui s'applique à la première variable rencontrée dans la chaîne (c'est la variable qui est le plus "à gauche" possible).

Théorème 1. Soit w une chaîne de L(G) . A toute suite de dérivations **S** => w correspond une unique suite de dérivations à gauche ayant le même arbre de dérivation.

Preuve. On associe son arbre de dérivation à la suite de dérivations donnée. Il suffit de parcourir cet arbre de dérivation en suivant les branches prioritairement le plus loin possible vers le bas, ensuite seulement de la gauche vers la droite ; pendant ce parcours, en repérant le premier passage à chaque nœud de l'arbre de dérivation, on obtient une suite de dérivations à gauche.

Cette façon de parcourir un arbre s'appelle ***"en profondeur d'abord"*** : on descend toujours autant que possible, en commençant à gauche ; on ne remonte que le minimum nécessaire pour pouvoir redescendre sur une branche située plus à droite. Un parcours ***"en largeur d'abord"*** se fait, lui, en allant d'abord de gauche à droite, ensuite de haut en bas.

Exercice 1. Ecrire la suite de dérivations à gauche associée à la suite de dérivations donnée dans l'exemple ci-dessus (vérifier le résultat : il faut obtenir le même arbre de dérivation).

Remarque 2. Des suites de dérivations différentes peuvent produire la même chaîne de L(G) , tout en étant associées à des arbres de dérivation différents.

Exercice 2. Toujours avec la grammaire G de l'exemple, trouver une autre suite de dérivations à gauche pour la chaîne ababa . Elle correspondra donc nécessairement à un autre arbre de dérivation ; tracer cet arbre de dérivation et comparer au précédent.

4. Grammaires régulières.

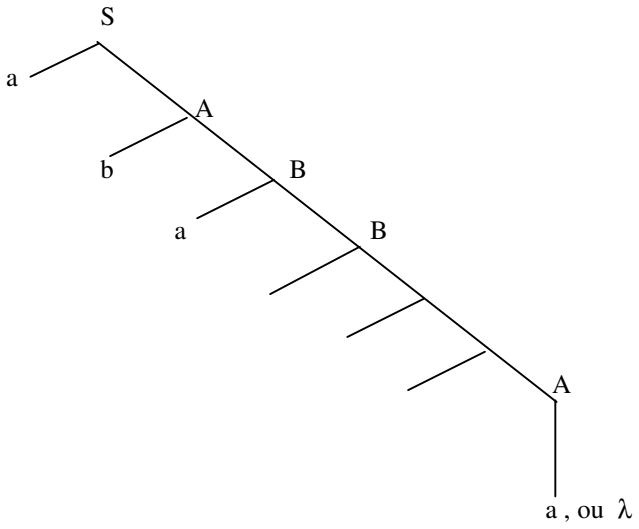
Définition. Une grammaire algébrique est dite *régulière* si toutes ses règles sont de l'un des types suivants :

- $A \rightarrow aB$
- $A \rightarrow a$ où $A, B \in V$ et $a \in \Sigma$
- $A \rightarrow \lambda$

Il en découle que les chaînes qui dérivent de l'axiome contiennent au plus une variable, et cette variable est nécessairement en fin de chaîne (en « suffixe »). Plus précisément, ces chaînes sont de l'une des formes suivantes :

- $a_1\, a_2\, a_3 \ldots a_n\, A$ une seule variable, précédée par un "préfixe terminal" $a_1\, a_2\, a_3 \ldots a_n \in \Sigma^+$
- $a_1\, a_2\, a_3 \ldots a_n$ chaîne terminale (obtenue après application d'une règle de type 2 ou 3)
- λ ce cas n'est possible que s'il y a, pour l'axiome, la règle vide $S \rightarrow \lambda$

Les arbres de dérivation associés aux suites de dérivation ont l'aspect d'un « peigne » (symboles et variables sont indiqués à titre d'exemple sur la figure suivante) :



Le théorème suivant met en relation les langages réguliers (que nous avons vus au chapitre précédent) et les grammaires régulières. Il sera difficile à prouver : plus précisément, la démarche suivie sera surtout très longue (mais très belle aussi).

Théorème 2. Tout langage régulier peut être engendré par une grammaire régulière. Réciproquement, si G une grammaire régulière, alors L(G) est un langage régulier.

Exercice 3. G : $S \rightarrow abSA \mid \lambda$

$A \rightarrow Aa \mid \lambda$

n'est pas une grammaire régulière. Mais…

1) Prouver que $L(G) = \{\lambda\} \cup (ab)^+.a^*$.
   Vérifier aussi que L(G) est différent de $(ab)^*.a^*$
   Le langage L(G) est donc un langage régulier. D'après le théorème 2 précédent, il doit exister une grammaire régulière qui l'engendre.
2) Trouver une grammaire régulière G_1 telle que $L(G_1) = L(G)$.

Une remarque sur la forme des règles d'une grammaire régulière.

On a vu (cf. la définition) que des règles du type $A \rightarrow a$ peuvent figurer dans une grammaire régulière. De telles règles peuvent être éliminées, à condition d'introduire une nouvelle variable dans la grammaire : il suffit en effet de remplacer la règle $A \rightarrow a$ par le groupe des deux règles suivantes

$A \rightarrow a.A_1$ (où A_1 désigne ici la nouvelle variable introduite)
$A_1 \rightarrow \lambda$

Cette opération est un exemple simple de transformation d'une grammaire en une autre grammaire qui lui est *équivalente* : le langage engendré reste inchangé.

Nous avons ainsi établi la proposition suivante.

Proposition. Toute grammaire régulière est équivalente à une grammaire régulière dont les règles sont du type $A \to aB$ ou $A \to \lambda$

Exemple. La grammaire G : $S \to bA \mid b \mid \lambda$
$A \to aA \mid aS \mid a$

est équivalente à G' : $S \to bA \mid bZ \mid \lambda$
$A \to aA \mid aS \mid aZ$
$Z \to \lambda$ (Z est ici la troisième variable de G')

Exercice 4. Décrire le langage engendré par ces grammaires, à l'aide d'une propriété caractérisant ses chaînes.

5. Ambiguïté. Graphe orienté d'une grammaire algébrique.

Dans la suite du cours, sauf mention du contraire, nous ne considérerons plus que des dérivations *à gauche* : en lisant une chaîne de gauche à droite, c'est toujours la première variable rencontrée qui est réécrite. Nous avons vu en effet (cf. théorème 1) que toute chaîne w de L(G) peut être obtenue à partir de l'axiome par une suite de dérivations à gauche.

Définition. Une grammaire G est dite *ambiguë* s'il existe dans L(G) une chaîne w que l'on peut obtenir par deux suites différentes de dérivations à gauche.

Remarque. L'ambiguïté est une propriété qui se rapporte à une grammaire, et non à un langage.

Il n'est pas difficile de trouver un exemple de langage engendré par une grammaire non ambiguë et par une (autre) grammaire qui, elle, est ambiguë : il suffit de partir d'une (petite) grammaire G dont on est certain qu'elle est non ambiguë, puis de la modifier en "doublant" une variable :

G : $S \to aA$
$A \to \lambda$

G' : $S \to aA \mid aB$
$A \to \lambda$
$B \to \lambda$

Exemple. L'exercice 2 (cf. §3.) permet en fait de prouver que la grammaire G considérée est ambiguë.

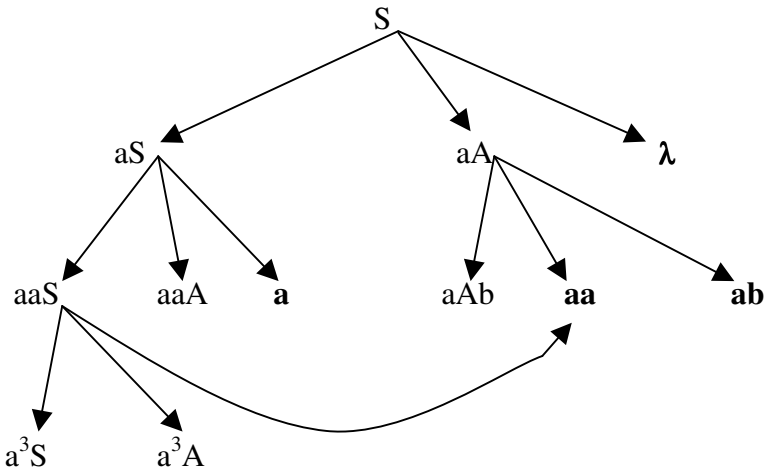
Définition. Une grammaire G étant donnée, on peut lui associer un *graphe orienté* (il s'agit d'un graphe infini, en général) :

- les *sommets* de ce graphe sont toutes les chaînes que l'on obtient à partir de l'axiome par des dérivations à gauche
- les *flèches* issues d'un sommet donné sont les règles de grammaire applicables à la première variable de la chaîne correspondant à ce sommet.

Exemple. Soit G : $S \rightarrow aS \mid aA \mid \lambda$

$A \rightarrow Ab \mid a \mid b$

Le graphe orienté de cette grammaire commence ainsi (il est infini, en fait) :



Sur cette partie du graphe orienté, l'ambiguïté de G est déjà mise en évidence puisque on y voit deux chemins différents qui mènent de S à la chaîne aa , chaîne qui est terminale et appartient donc au langage engendré.

De manière générale :

- Dans le graphe orienté de la grammaire, un arbre de dérivation (cf. §3.) est représenté par un *chemin* partant de l'axiome et aboutissant à un élément de L(G) . En effet, un arbre de dérivation correspond à une (unique) suite de dérivations à gauche, cf. théorème 1.
- La grammaire est ambiguë si et seulement si son graphe orienté contient des chemins différents issus de l'axiome et aboutissant à une même chaîne de L(G).

LE LEMME DE L'ETOILE (en anglais : « pumping lemma »)

Le problème : un langage L étant donné, comment savoir s'il peut être engendré par une grammaire régulière ?

Une réponse : le *lemme de l'Etoile* donne une condition que les chaînes de L doivent nécessairement satisfaire s'il y a une telle grammaire.

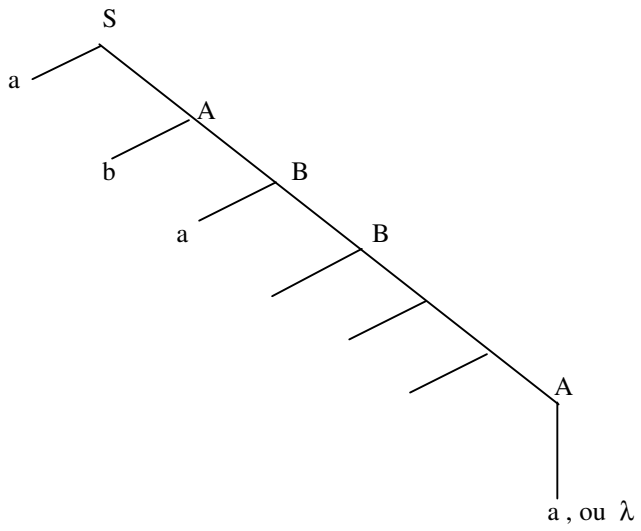
Cette condition nécessaire sera ensuite exploitée pour prouver que, pour certains langages, il n'existe pas de grammaire régulière : il suffira de produire des chaînes servant de « contre-exemple », c'est-à-dire des exemples de chaînes du langage qui ne vérifient pas la condition énoncée par le lemme de l'Etoile.

1. Le lemme de l'Etoile pour les grammaires régulières.

Rappelons que pour une grammaire G *régulière*, les règles sont de l'une des formes suivantes :

- $A \to aB$ où $a \in \Sigma$ et $B \in V$
- $A \to a$ où $a \in \Sigma$
- $A \to \lambda$

et si w est un élément du langage engendré L(G) , il y a un arbre de dérivation de la forme particulière suivante :

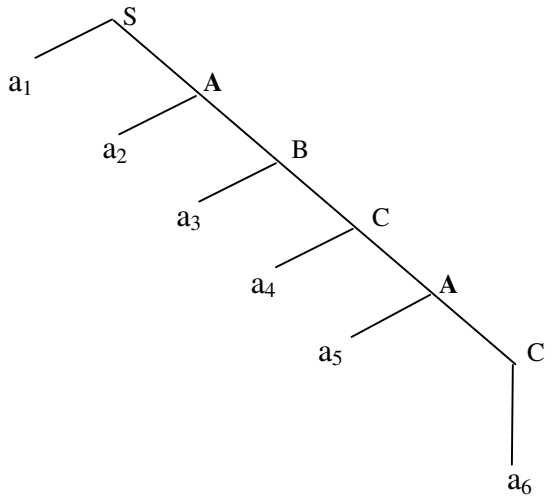


où w est la chaîne formée par les symboles figurant aux extrémités des branches (symboles et variables sont indiqués à titre d'exemple).

Faisons une remarque évidente, mais décisive pour la suite : si la longueur de la chaîne w est strictement supérieure au nombre de variables de la grammaire G , alors il y a une variable (au moins) qui figure deux fois dans l'arbre de dérivation.

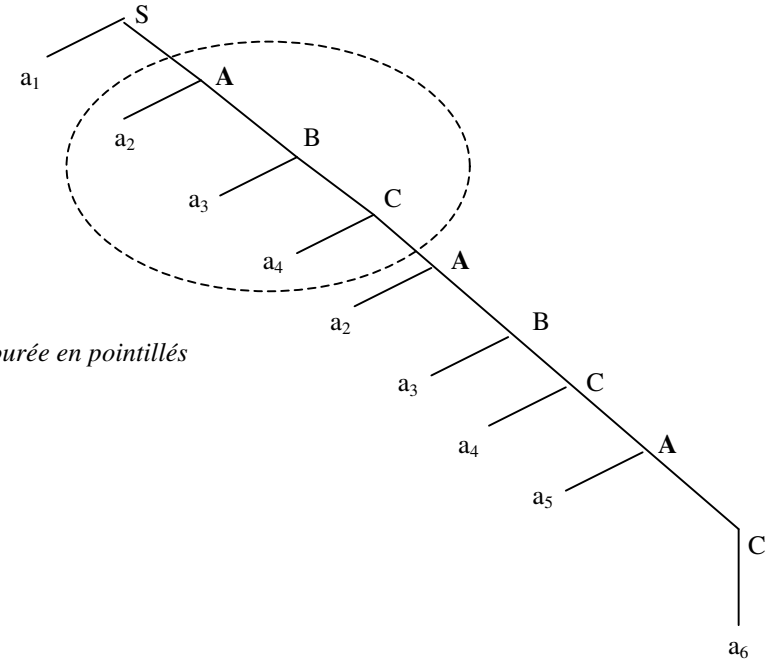
Exemple. La figure suivante représente un arbre de dérivation pour la chaîne $w = a_1a_2a_3a_4a_5a_6$ de L(G) , et cette chaîne est supposée être de longueur strictement supérieure au nombre de variables de G . La variable A figure deux fois dans l'arbre de dérivation.

Lors de la première occurrence de A, on a appliqué une certaine règle de dérivation, disons $A \rightarrow a_2B$; puis une succession de règles diverses ; à la deuxième occurrence de la variable A, on a appliqué une règle $A \rightarrow a_5C$; puis encore diverses règles.



Dans cette situation, nous pouvons construire de nouvelles chaînes qui devront certainement, elles aussi, appartenir au langage L(G), pour la bonne raison qu'elles sont produites en se servant uniquement de règles qui ont été utilisées pour produire w , donc des règles qui sont dans la grammaire G . La procédure est la suivante : jusqu'à la *deuxième* occurrence de la variable répétée, on applique les mêmes règles que celles qui ont servi pour produire w ; à ce moment-là, on réutilise la règle qui avait déjà été appliquée lors de la *première* occurrence, et celles qui ont suivi.

Sur notre exemple, au lieu d'appliquer $A \rightarrow a_5C$, on réutilise la règle $A \rightarrow a_2B$ et celles qui suivent. Nous obtenons ainsi une nouvelle chaîne $w' = a_1a_2a_3a_4a_2a_3a_4a_5a_6$ qui appartient également à L(G).



La partie de l'arbre entourée en pointillés a été dupliquée

Bien entendu, la duplication de ce tronçon de l'arbre de dérivation peut être recommencée autant de fois que l'on veut : la chaîne $a_1\, a_2\, a_3\, a_4\, a_2\, a_3\, a_4\, a_2\, a_3\, a_4\, a_5\, a_6$ et, plus généralement, toutes les chaînes qui sont de la forme $a_1(a_2a_3a_4)^i.a_5a_6$ appartiennent nécessairement au langage $L(G)$.

En outre, il est évident que si K désigne le nombre de variables dans la grammaire, la répétition d'une variable se produira déjà parmi les $K+1$ premiers noeuds de l'arbre (la racine S est comptée comme le premier noeud).

Nous obtenons ainsi le résultat suivant :

Proposition. Soit G une grammaire régulière. On note K le nombre de ses variables et $L(G)$ le langage engendré.
Alors, toute chaîne w de $L(G)$ dont la longueur est strictement supérieure à K peut s'écrire sous la forme $w = u.x.v$
où : x est une chaîne non vide
la longueur de la chaîne $u.x$ est inférieure ou égale à K
la chaîne $u.x^i.v$ appartient à $L(G)$ pour tout entier i .

Rappelons le théorème 2 (que nous prouverons dans le chapitre sur les automates) : pour tout langage régulier L , il existe une grammaire régulière G telle que $L = L(G)$. De cette grammaire dont on ignore à peu près tout, nous savons au moins qu'elle a un nombre fini de variables. Nous ignorons quel est ce nombre, mais nous pouvons maintenant appliquer la proposition précédente aux chaînes du langage régulier considéré.

Voici donc la version définitive du « critère de régularité » qui découle des considérations et de la proposition précédentes :

« Lemme de l'Etoile » pour les langages réguliers.

Soit L un langage régulier. Il existe une constante K telle que toute chaîne w de L dont la longueur est strictement supérieure à K peut s'écrire sous la forme $w = u.x.v$

où : x est une chaîne non vide,

la longueur de $u.x$ est inférieure ou égale à K ,

la chaîne $u.x^i.v$ appartient au langage L pour tout entier i .

Voyons maintenant un exemple d'utilisation de ce critère, pour prouver que certains langages ne sont pas réguliers.

2. Un exemple d'application du lemme de l'Etoile.

Proposition. Soit l'alphabet $\Sigma = \{a, b\}$.
Le langage $L = \{a^n.b^n : n \geq 0\}$ n'est pas un langage régulier.

Preuve de la proposition. On raisonne par l'absurde. Supposons que L est un langage régulier. D'après le Lemme de l'Etoile, il doit exister une longueur K telle que toute

chaîne w de L dont la longueur est supérieure à K contienne au moins une séquence x (non vide) qu'on peut dupliquer dans la chaîne sans sortir du langage. Autrement dit, il est possible d'écrire w sous la forme d'une concaténation $w = uxv$ ayant la propriété que la chaîne $uxxv$ est encore dans L. De plus, la séquence x se trouve parmi les K premiers symboles de w.

Or, *pour n'importe quel* nombre K, nous pouvons trouver une chaîne w qui fait usage de « contre-exemple » : il nous suffit de considérer la chaîne $w = a^K.b^K$. Sa longueur est $2K$, donc supérieure à K, et comme la séquence x doit se trouver parmi les K premiers symboles de w, elle est formée uniquement de a, c'est-à-dire qu'elle est de la forme $x = a^p$ (avec $p \geq 1$) :

$$w = \underbrace{a\ldots}_{u}\ \underbrace{a^p}_{x}\ \underbrace{\ldots a.\, b^K}_{v}$$

Mais alors la chaîne $uxxv = a^{K+p}.b^K$, et cette chaîne n'est pas dans le langage. Le lemme de l'Etoile est contredit.

On en conclut que L n'est pas un langage régulier.

Définition. Un *palindrome* sur un alphabet Σ donné est une chaîne de symboles de Σ qui peut être lue indifféremment de gauche à droite ou de droite à gauche.

Quelques exemples de palindromes, sur $\Sigma = \{a, b\}$: aabaa, abba, abababa, aaaa.

Exercice. Prouver que le langage des palindromes sur $\Sigma = \{a, b\}$ n'est pas un langage régulier.

Note. Il existe une version « perfectionnée » de Lemme de l'Etoile pour les langages engendrés par des grammaires algébriques.

ANALYSE SYNTAXIQUE (en anglais : « parsing »)

On se donne une grammaire algébrique G.

Le problème : décrire un algorithme permettant de déterminer si une chaîne donnée $w \in \Sigma^*$ appartient, ou non, à $L(G)$.

Deux stratégies sont possibles :

- avec une analyse syntaxique *descendante*, on part de l'axiome, et on recherche la chaîne w
- avec une analyse syntaxique *ascendante*, on part au contraire de la chaîne w, et on cherche à rejoindre l'axiome.

Nous verrons de manière détaillée l'analyse syntaxique descendante. Celle-ci peut se faire selon deux types de parcours :

- en largeur d'abord
- en profondeur d'abord

1. Test sur le préfixe terminal dans les analyses syntaxiques descendantes.

Définition. Le *préfixe terminal* d'une chaîne $u \in (\Sigma \cup V)^*$ est la sous-chaîne des symboles terminaux qui précédent la première variable figurant dans u.

Remarque. Lorsqu'on procède à une dérivation, le préfixe terminal se trouve soit inchangé soit prolongé.

Cette remarque simple est à la base du test qui se fera lors de l'analyse syntaxique : si le préfixe terminal d'une chaîne $u \in (\Sigma \cup V)^*$ n'est pas identique au début de la chaîne analysée w, il est impossible que w dérive de u.

Le test : tester une chaîne u consiste à comparer son préfixe terminal avec le début de la chaîne analysée w.

Noter qu'en informatique, la donnée d'une chaîne de symboles est toujours suivie d'un signe de fin de chaîne : espace, guillemet, ou tout autre caractère spécifique servant à signifier la fin. Dans le cas où u est complètement terminale (autrement dit dans Σ^*), on arrive au signe de fin de chaîne (puisqu'on n'a pas trouvé de variable), et on teste donc l'égalité des chaînes : $u = w$ oui ou non.

Remarque. Par une dérivation à gauche, en appliquant une règle de grammaire qui commence par un symbole terminal, on est certain que le préfixe terminal sera allongé ; le test sera donc plus efficace. Mieux : la complexité de l'analyse syntaxique sera contrôlée si toutes les règles commencent par un symbole terminal. C'est pourquoi une étape préliminaire consiste à transformer dans ce but la grammaire donnée. Nous verrons comment procéder à ce type de transformation dans un chapitre ultérieur.

2. Analyse syntaxique descendante en largeur d'abord.

L'algorithme d'analyse.

- Les règles de G sont ordonnées (c'est-à-dire numérotées, mises dans une liste).
- Le test sur les chaînes : comparer leur préfixe terminal au début de la chaîne w à analyser.
- Une file sera créée, afin de mettre en attente les chaînes :
  - qui dérivent de l'axiome,
  - qui contiennent encore au moins une variable
  - et qui restent susceptibles d'aboutir à w par dérivation (cf. le test).

  Rappelons le principe d'une file : *" le premier entré est le premier sorti ".*

Les entrées : la grammaire (symboles, variables, règles), et la chaîne à analyser.

Création d'une file Q :

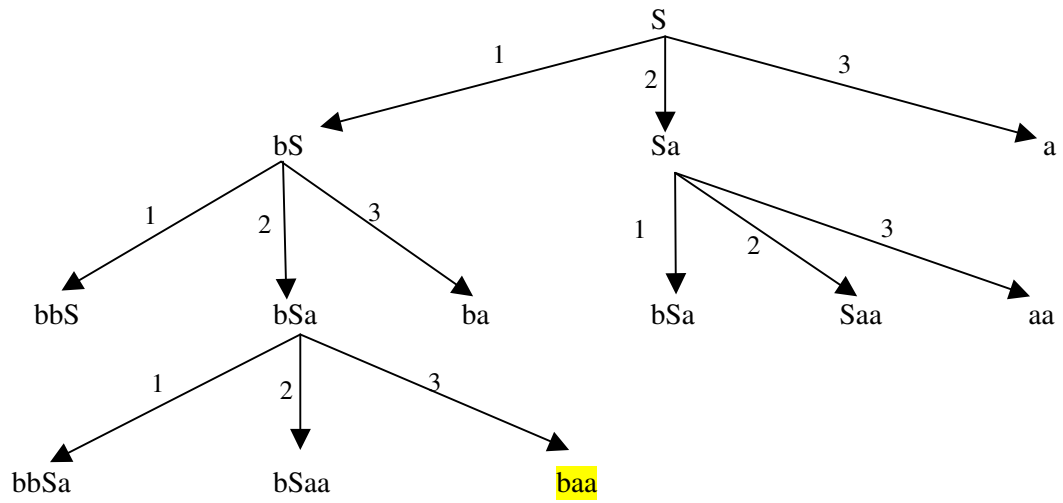
1. Initialiser la file : Q = [S]
2. Soit q le 1er élément de la file Q
  2.1. sortir q de la file
  2.2. appliquer successivement chacune des règles applicables à la 1ère variable de q et tester chaque chaîne ainsi obtenue :
  si la chaîne est non terminale et si le test est positif, l'enfiler dans Q
  2.3. retour à 2
3. Arrêt : quand le chaîne w est trouvée ou quand la file Q est vide au retour à 2.

Exemple. G : $S \rightarrow bS \mid Sa \mid a$ Numérotation de ces règles : 1 | 2 | 3

La chaîne à analyser est w = baa .

Un *arbre d'analyse* permet de représenter[1] à l'aide d'un schéma le déroulement de l'analyse : à partir de chacune des chaînes prélevées dans la file, des flèches pointent vers les chaînes que l'on obtient par une dérivation à gauche. Si le test sur la chaîne obtenue est négatif, l'exploration suivant cette branche est arrêtée puisque la chaîne n'est pas mise dans la file.

Remarquer qu'une même chaîne (ci-dessous, bSa par exemple) peut figurer à plusieurs endroits de l'arbre d'analyse : ceci signifie qu'elle a été examinée à plusieurs reprises. Elle peut même figurer, à un moment donné, en plusieurs exemplaires dans la file d'attente Q . Une telle situation, due à la forme de la grammaire, ralentit bien sûr l'analyse.



Exercice. Pour cette analyse syntaxique, suivre sur l'arbre l'ordre dans lequel les chaînes ont été examinées par l'algorithme ; préciser le contenu de la file Q .

Quelles sont les chaînes qui restent encore dans la file, au moment où l'analyse s'arrête ?

3. Analyse syntaxique descendante en profondeur d'abord.

L'algorithme d'analyse.

- Les règles de G sont ordonnées (mises dans une liste).
- Le test sur les chaînes : comparer leur préfixe terminal au début de la chaîne w à analyser.
- Une pile sera créée, afin de mettre en attente les chaînes :
  - qui dérivent de l'axiome,
  - qui contiennent encore au moins une variable
  - et qui restent susceptibles d'aboutir à w par dérivation (cf. le test).

  Rappelons le principe d'une pile : *" le dernier entré est le premier sorti "*.
- On empilera une chaîne q accompagnée d'un n° de règle applicable à sa première variable.
- Lorsqu'on dépilera un élément [q, i] , on cherchera s'il y a une règle suivante applicable à q (donc n° i+1), et dans ce cas on empilera [q, i+1] . S'il n'y a plus de règle suivante applicable à q , on dépilera un élément supplémentaire.
- L'analyse s'arrête si la pile est vide, ou si la chaîne w est trouvée ; dans ce dernier cas, la pile contient alors un chemin de dérivations à gauche de S à w .

Les entrées : la grammaire (symboles, variables, règles), et la chaîne à analyser.
Création d'une pile d'analyse P :
(Rappel : on empile une chaîne accompagnée d'un numéro de règle applicable à sa première variable).

1. Initialiser la pile P : P = [S, 1] .
Cette règle n°1 : S → q produit la chaîne q .

2. Tester la chaîne q .
Si le test sur q est positif, alors

2.1 - chercher la première règle applicable à la première variable dans q ,
disons q = uAv (où u est le préfixe terminal) , et n° i : A → y
- empiler [q, i]
- affecter q = uyv
- retour au test, en **2.**

sinon *(cas où le test est négatif)*

2.2 - dépiler l'élément en haut de la pile, disons [q', i']
- Si la règle suivante (n° i'+1) est encore applicable à q',
disons q'=u'A'v' et n° i'+1 : A' → z

alors
- empiler [q', i'+1]
- affecter à q la chaîne obtenue avec cette règle : q = u'zv'
- retour à **2.**

sinon, retour à **2.2.**

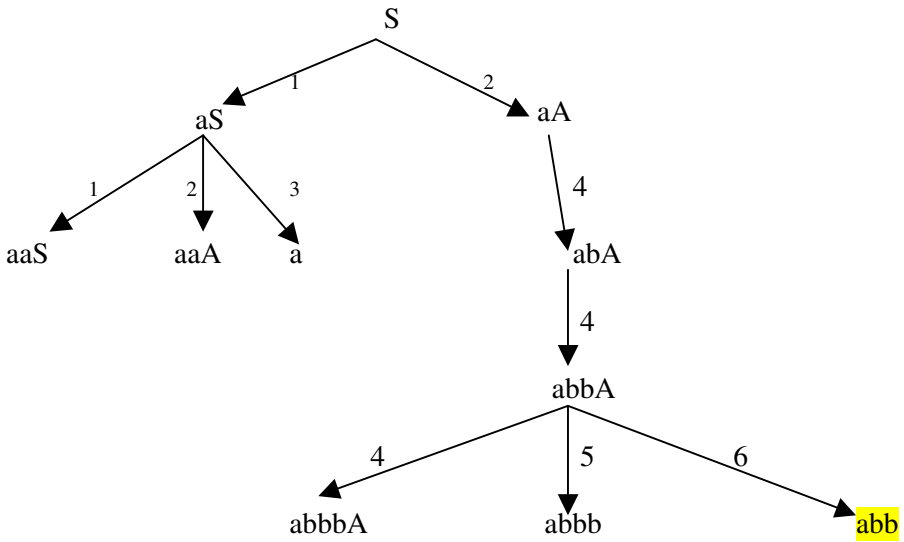
3. Arrêt : si w est trouvée, ou si la pile est vide au retour à 2.2.

Là encore, on peut représenter le déroulement de l'algorithme à l'aide d'un *arbre d'analyse* : cette fois, il sera tracé prioritairement de haut en bas, ensuite seulement de la gauche vers la droite. Les opérations d'empilement et de dépilement se voient aisément sur l'arbre : il suffit de tracer un chemin qui « contourne » l'arbre en suivant toutes ses branches : il part de S (à gauche de l'arbre, sur le dessin suivant) et s'arrête à la chaîne analysée w si elle est trouvée. Si l'analyse ne trouve pas w , le chemin contourne entièrement l'arbre et aboutit à S .

Exemple. G : $S \to aS \mid aA \mid \lambda$ numérotées : 1 | 2 | 3

$A \to bA \mid b \mid \lambda$ 4 | 5 | 6

La chaîne à analyser est w = abb



Au moment où cette analyse s'arrête (elle s'arrête parce qu'on a trouvé w), le contenu de la pile P est :

[abbA, 6] *(le haut de la pile)*
[abA, 4]
[aA, 4]
[S, 2]

On retrouve donc dans la pile un chemin joignant S à w , c'est-à-dire toute une suite de dérivations à gauche :

$$S \Rightarrow aA \Rightarrow abA \Rightarrow abbA \Rightarrow w$$

Ceci est un aspect intéressant de l'analyse en profondeur.

Remarque. L'analyse syntaxique peut être longue, ou même inopérante, selon la forme des règles de grammaire. Les règles qui commencent par une variable, et particulièrement celles qui sont de la forme $A \to Au$ (où u est une chaîne quelconque), dites *directement récursives à gauche*, peuvent empêcher l'analyse syntaxique d'aboutir. Mais nous verrons ultérieurement des algorithmes qui permettent de transformer la grammaire en une grammaire équivalente pour laquelle toutes les règles (non vides) commencent par un symbole terminal.

4. Note sur les analyses ascendantes.

Le principe est de partir de la chaîne w à analyser et de chercher à remonter à S.
Là encore, on peut faire l'exploration en largeur d'abord ou bien en profondeur d'abord.

Une règle $A \to u$ donne des dérivations de la forme $pAq \Rightarrow puq$. Il faut donc pouvoir reconnaître la séquence u dans une chaîne, afin de pouvoir la remplacer par la variable A. Pour cela, on pratique le test de la manière suivante.

Une chaîne $w \in \sum^*$ étant donnée, on découpe w en deux chaînes concaténées $w = y.z$. Ce découpage est initialisé avec $y = \lambda$ et $z = w$, puis il progressera vers la droite : on allongera y et raccourcira z d'un symbole à la fois. A chaque fois, les règles de grammaire sont successivement comparées à la fin de la chaîne y.

S'il y a une règle $A \to u$ telle que u soit un suffixe de y, disons $y = y_1.u$:

1) remplacer u par A. On obtient ainsi la chaîne $y_1.A.z$, qui est affectée à w.
2) retour au découpage de w. Mais comme cette chaîne w contient maintenant des variables, le découpage se fait dorénavant avec la condition supplémentaire que $z \in \sum^*$; autrement dit, le découpage est réinitialisé juste après la dernière variable de w (ceci afin de gagner du temps de calcul).

Noter qu'on obtient ainsi une suite de dérivations *à droite*, qu'il suffit d'ailleurs d'avoir gardées en mémoire dans une pile pour reconstituer une dérivation (à droite) $S \Rightarrow w$.

L'avantage de la méthode ascendante est la rapidité : la chaîne est raccourcie à chaque substitution, à condition toutefois qu'il n'y ait pas de règles du type « enchaînement de variables » $A \to B$; mais nous verrons comment éliminer ce type de par une transformation de la grammaire.

AUTOMATES

Soit L un langage sur un alphabet de symboles Σ .

Le problème : peut-on décrire une petite "machine" qui acceptera ou rejettera une chaîne w de Σ^* selon qu'elle appartient, ou non, au langage L ? Un modèle du genre est un automate distributeur de café : si l'appoint en monnaie est correct, la machine répond en versant un café ; dans le cas contraire, elle reste en attente (ou donne un message d'erreur)…

Un automate (théorique) sera décrit en précisant :

- son alphabet de symboles (les pièces de monnaie acceptées),
- ses états initiaux,
- ses états finaux (café, avec ou sans sucre, chocolat, …),
- les états intermédiaires dans lesquels il peut se trouver (pendant qu'on fait l'appoint, par exemple),
- ainsi que les transitions, c'est-à-dire comment on passe d'un état à un autre.

1. Définitions.

Définition. Un *automate* est défini par un 5-uplet $A = (\Sigma , E, E_o , F, \delta)$, où

Σ est l'ensemble fini des *symboles*

E est un ensemble fini : l'ensemble des *états*

$E_o \subset E$ est le sous-ensemble des *états initiaux*

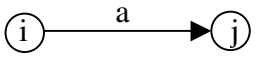
$F \subset S$ est le sous-ensemble des *états finaux*

δ est un ensemble fini de *transitions* :

une *transition* est un triplet (i, a, j) , où i et j sont des états et a est un symbole.

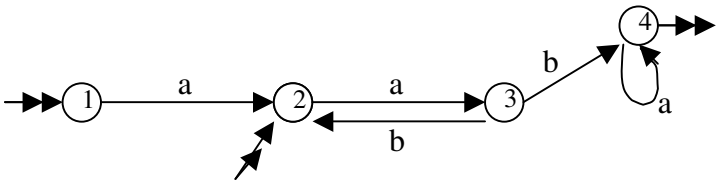
Représentation graphique d'un automate.

Une transition (i, a, j) se représente par une flèche de i à j , munie d'une *étiquette* a :



Un automate se représente donc à l'aide d'un graphe orienté, sur lequel il faut toutefois indiquer les états particuliers que sont les entrées et les sorties. Les états initiaux sont indiqués par une double flèche entrante ; pour les états finaux, une double flèche sortante.

Exemple.

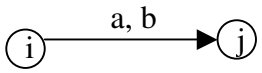


Pour cet exemple, l'ensemble des symboles est $\Sigma = \{a, b\}$.
L'ensemble des états est $E = \{1, 2, 3, 4\}$; parmi ceux-ci, il y a deux états initiaux et un état final : $E_o = \{1, 2\}$ et $F = \{4\}$. Les transitions sont décrites dans l'ensemble

$$\delta = \{(1,a,2), (2,a,3), (3,b,2), (3,b,4), (4,a,4) \}$$

Pour simplifier les dessins, on convient de représenter deux transitions entre les mêmes états (i,a,j) et (i,b,j) par une seule flèche, munie des deux étiquettes.

Il faut donc comprendre " a ou b ", en lisant l'étiquette suivante :



2. Langage reconnu par un automate.

Définition. Une chaîne $w \in \Sigma^*$ est *reconnue* par un automate $\mathcal{A}$ s'il y a un cheminement, c'est-à-dire une suite de transitions, partant d'un état initial, aboutissant à un état final, et dont la suite des étiquettes est w .

Définition. Le *langage reconnu* par un automate $\mathcal{A}$ est l'ensemble de toutes les chaînes reconnues. Il est noté $L(\mathcal{A})$.

Exercice.

1) Parmi les chaînes aabaa, aabab, baba, ab, abaaaa , déterminer lesquelles sont reconnues par l'automate $\mathcal{A}$ représenté au §1.

2) Déterminer le langage $L(\mathcal{A})$ reconnu.

Remarque. L'automate $\mathcal{A}$ représenté au §1 est « non-déterministe » : quand on est dans l'état 3 et que le symbole b est lu, il est possible d'aller en 2 ou bien en 4 ; ce choix est fait au hasard. De plus, il y a le choix entre deux entrées possibles (les états 1 et 2).

3. Automates déterministes, automates déterministes complets.

Définition. Un *automate déterministe (AD)* est un automate :

- avec un unique état initial
- à partir de chaque état, il y a au plus une transition d'étiquette donnée.

Définition. Un *automate déterministe complet (ADC)* est un automate déterministe pour lequel il y a, à partir de chaque état, une et une seule transition pour chaque étiquette possible de Σ .

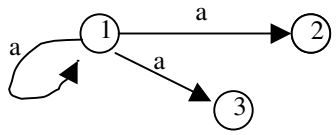
L'intérêt d'un ADC est qu'en lisant une chaîne (quelconque) w de Σ^* , il lui correspond un parcours bien défini à partir de l'état initial, et la progression dans l'automate ne se trouve jamais bloquée : la chaîne peut toujours être lue jusqu'au bout. La chaîne w est reconnue si et seulement si on aboutit à l'un des états « de sortie », c'est-à-dire à un état final.

Théorème 3. Tout automate peut être transformé en un automate déterministe reconnaissant le même langage.

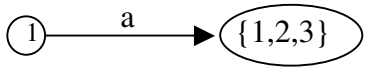
Preuve du théorème 3.

L'idée est simple : il faut regrouper les états auxquels on peut arriver en lisant un même symbole.

Exemple



Si on est à l'état 1 et lit le symbole a , on va vers « 1 ou 2 ou 3 » On groupe donc tous les états possibles d'arrivée en un seul, ce qui crée un nouvel état « 1 ou 2 ou 3 » :



Bien entendu, cette construction doit se faire « de proche en proche », à partir des états initiaux, qui sont les premiers états qu'il faut regrouper pour obtenir une entrée unique. C'est donc une *construction récursive*, et algorithmique.

Soit $A = (\Sigma , E, E_o, F, \delta)$ l'automate initial donné.

Le nouvel automate, déterministe, sera noté $A' = (\Sigma , E', \{s'_o\}, F', \delta')$. Les états de ce nouvel automate sont des groupes d'états de l'ancien automate, c'est-à-dire des sous-ensembles de l'ensemble E . L'état initial s'_o est unique.

L'algorithme.

- initialiser : $s'_o = E_o$ *(s'_o est l'ensemble de tous les états initiaux de A. C'est l'état initial de A')*

  $E' = \{s'_o\}$ *(initialisation de la construction de l'ensemble des états E')*

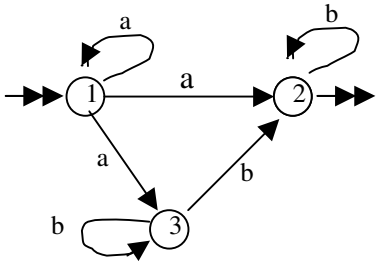
- répéter, jusqu'à ce que l'ensemble E' soit stationnaire (c'est-à-dire que plus aucun élément nouveau ne s'y rajoute) :

  pour chaque état $A' \in E'$ venant d'être construit, et pour chaque symbole $a \in \Sigma$,

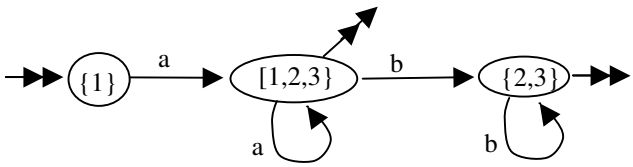
  - considérer (dans A) toutes les transitions d'étiquette a issues d'un état $A \in A'$, et regrouper leurs états d'arrivée
  - si ce groupe d'états n'est pas encore un élément de E', on crée ce nouvel état en le rajoutant à l'ensemble E'
  - rajouter la transition d'étiquette a issue de A' vers cet état.

- définir les états finaux de A' : ce sont tous les états qui contiennent au moins un état final de A .

Exercice. Construire l'AD à partir de l'automate suivant



Réponse :



Note. On pourra omettre l'écriture des accolades, pour alléger la notation.

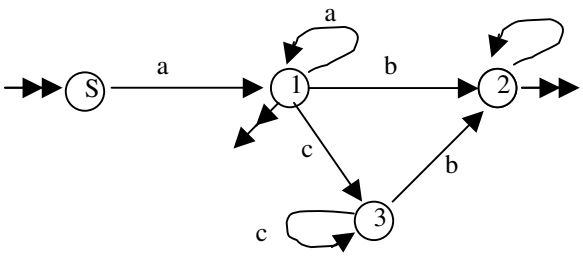
Théorème 4. Tout automate déterministe peut être transformé en un automate déterministe complet reconnaissant le même langage.

Preuve du théorème 4.

L'opération est élémentaire : si l'AD n'est pas déjà complet,

- lui rajouter un nouvel état (« état poubelle »)
- rajouter les transitions d'étiquettes manquantes en les dirigeant toutes vers cet état poubelle P ; ne pas oublier les transitions de P lui-même vers P .

Exercice. compléter l'AD suivant



Théorème 5. Soit L un langage reconnu par un automate.
Alors le langage complémentaire $\Sigma^* - L$ est aussi reconnu par un automate.

Preuve du théorème 5. Partir d'un automate $\mathcal{A}$ reconnaissant L ; le transformer en un AD, puis en un ADC . Rappelons qu'avec l'ADC, toutes les chaînes peuvent être lues jusqu'au bout : la lecture des chaînes de L fait arriver à un état final, alors que celle des chaînes n'appartenant pas à L fait aboutir à un état non final.

Sur la représentation graphique, il suffit maintenant d'ôter les sorties existantes, et de mettre des sorties aux états qui n'en avaient pas auparavant, pour obtenir un ADC reconnaissant le langage complémentaire du langage L .

4. Automates et grammaires régulières.

Théorème 6. On peut associer à toute grammaire *régulière* G un automate reconnaissant le langage L(G) . Donc aussi un ADC reconnaissant L(G) .

Réciproquement, on peut associer à tout automate $\mathcal{A}$ ayant *un unique état initial* une grammaire régulière qui produit le langage reconnu par $\mathcal{A}$.

Remarque. A partir d'un automate quelconque, on sait obtenir un AD reconnaissant le même langage ; or celui-ci a bien un seul état initial.

Preuve du théorème 6.

1) La grammaire régulière G est, si nécessaire, modifiée afin que toutes ses règles soient de l'un des types suivants (cf. Proposition, p.12) :

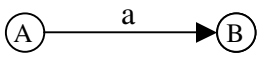
- $A \to aB$ (où $a \in \Sigma$ et $A, B \in V$)
- $A \to \lambda$

2) La correspondance se fait :

- entre les variables de la grammaire G et les états de l'automate A
- entre les règle de grammaire et les transitions de l'automate

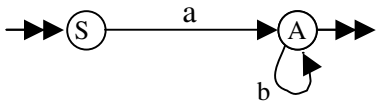
l'axiome S de la grammaire	:	l'(unique) état initial de l'automate
les variables	:	les états
les règles	:	les transitions, ou les sorties :

$A \to aB$:



$A \to \lambda$:

Exemple. G : $S \to aA$
$A \to bA \mid \lambda$



Corollaire. Les langages reconnus par les automates sont les langages engendrés par les grammaires régulières.

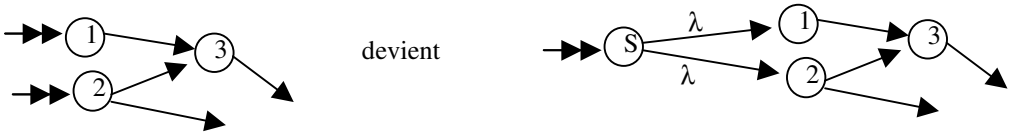
5. Les λ–automates. Preuve du théorème 2.

Il s'agit d'une légère généralisation de la notion d'automate. Son intérêt est technique : elle nous permet de faire les « bricolages » à partir d'automates dont nous aurons besoin par la suite (jonction, réunion,…).

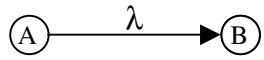
Définition. Un λ–*automate* se définit de manière analogue à un automate, néanmoins :

- on autorise des transitions d'étiquette vide (l'étiquette est alors notée λ)
- on exige qu'il n'y ait qu'un seul état initial

Remarque. Si des transitions d'étiquette vide sont permises, la deuxième condition n'est plus du tout contraignante ! s'il y a plusieurs états initiaux, il suffit de banaliser ces états et de rajouter un nouvel état, qui devient l'unique état initial, avec des transitions d'étiquette vide adéquates :



L'unicité de l'état initial est une condition indispensable si l'on veut progresser vers la construction d'une grammaire. Du point de vue des grammaires, d'ailleurs, une transition d'étiquette vide corespond très exactement à une règle d' « enchaînement de variables » (les états A et B du λ–automate correspondent à des variables A et B de la grammaire) :

$A \to B$: 

Les grammaires naturellement associées aux λ–automates ne sont donc pas les grammaires régulières : on autorise, de plus, des règles du type « enchaînement de variables » .

Néanmoins :

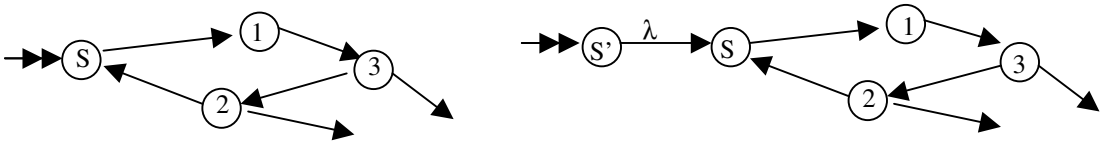
Théorème 7. Tout λ–automate peut être transformé en un automate, avec un état initial unique et non récursif.

L'état initial est dit *non récursif* si on n'y passe qu'une seule fois (à l'entrée dans l'automate) ; autrement dit, il n'y a pas de cheminement qui y revient en boucle fermée.

Preuve du théorème 7.

Un λ–automate étant donné,

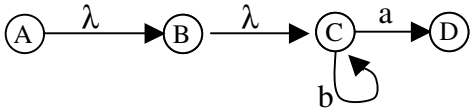
1) on commence par supprimer, si nécessaire, la récursivité de l'état initial en rajoutant un nouvel état initial et banalisant l'ancien, avec une transition d'étiquette vide de l'un à l'autre :



2) il faut maintenant supprimer les transitions d'étiquette vide, ce qui ne peut se faire qu'en ajoutant de nouvelles transitions afin de compenser ces suppressions (le langage reconnu doit rester inchangé). Pour cela, il faut :

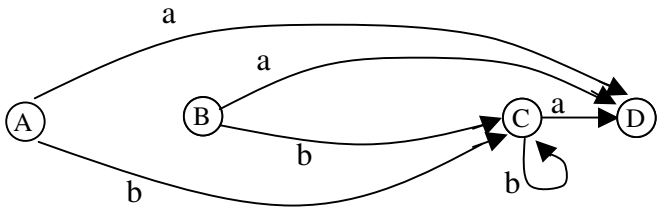
- repérer tous les chemins qui sont constitués d'une succession de transitions d'étiquette vide, suivie d'une transition d'étiquette non vide.
- remplacer chacun de ces chemins par une nouvelle transition « court-circuitant » la suite des transitions d'étiquette vide ; elle est bien sûr étiquetée avec le même symbole que la dernière transition du chemin.

Exemple :

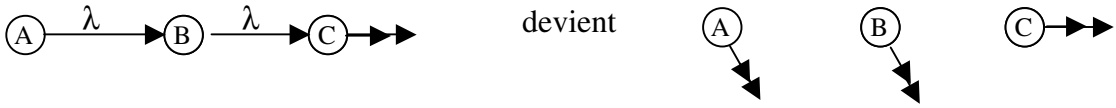


Nous avons ici deux chemins du type voulu qui sont issus de l'état A (l'un aboutit à D, l'autre aboutit à C après avoir suivi la transition d'étiquette b) ; nous avons aussi deux chemins analogues issus de l'état B .

Nous obtenons les transitions suivantes :



- Attention à ne pas oublier les sorties : si des transitions d'étiquettes vides aboutissent à un état final, elles ne peuvent être supprimées que si leur état de départ devient un état final.



Théorème 8. Sont reconnus par un λ–automate les langages suivants :
- le langage {λ}
- les langages de la forme {a} , avec a ϵ Σ
- la réunion de deux langages reconnus par un λ–automate
- la concaténation de deux langages reconnus par un λ–automate
- l'étoile de Kleene d'un langage reconnu par un λ–automate.

Donc, tout **langage régulier** est reconnu par un λ–automate.

Soit L un langage régulier. D'après le théorème 8, il est reconnu par un λ–automate, qui peut être transformé en un automate (et même un ADC) à un seul état initial, non récursif, d'après le théorème 7 ; à cet automate correspond une grammaire régulière (dont l'axiome sera d'ailleurs non récursif), d'après le théorème 6.

Nous avons ainsi obtenu une preuve du théorème 2 , que nous rappelons ici :

Théorème 2 (p.12). Tout langage régulier est engendré par une grammaire régulière.

Ce qui précède nous permet aussi de prouver la Proposition 2 , déjà énoncée p.7 :

Proposition 2 (p.7). Le complémentaire d'un langage régulier est aussi un langage régulier.
L'intersection de deux langages réguliers est aussi un langage régulier.

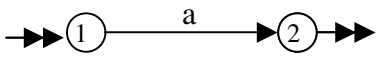
Preuve de la Proposition 2. Nous avons vu avec le théorème 5 (p.27) comment modifier un ADC afin de reconnaître le langage complémentaire.

Pour ce qui est de *l'intersection* de deux langages réguliers, il suffit de remarquer que l'intersection de deux ensembles est le complémentaire de la réunion des complémentaires !

Preuve du théorème 8.

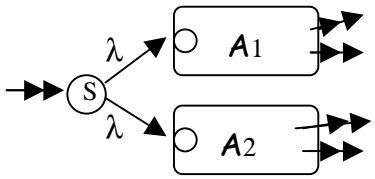
Il suffit d'associer un λ–automate adéquat à chacun des types de langages énumérés :

- langage {λ} : 

- les langages de la forme {a} : 

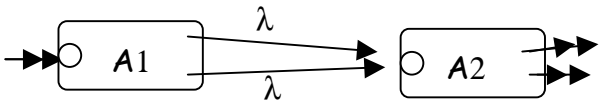
- la réunion de deux langages L_1 et L_2 reconnus par des λ–automates A_1 et A_2 :

banaliser l'état initial de chacun, rajouter un nouvel état initial relié à chacun par une transition d'étiquette vide. On peut ainsi entrer dans A_1 *ou* dans A_2 : une chaîne sera reconnue si elle appartient à L_1 ou à L_2 :



- la concaténation $L_1.L_2$ de deux langages L_1 et L_2 reconnus par des λ–automates A_1 et A_2 :

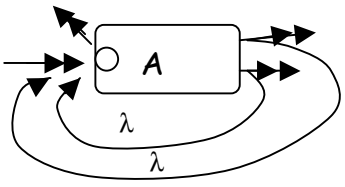
banaliser les états finaux de A_1 et l'état initial de A_2 , et ajouter des transitions d'étiquette vide des premiers vers le deuxième :



- l'étoile de Kleene d'un langage L reconnu par un λ–automate A :

en veillant à ce que le λ–automate A reconnaissant L ait son état initial non récursif (cf. théorème 7), rajouter des transitions d'étiquette vide allant des états finaux vers l'état initial.

Si la chaîne vide n'est pas dans L , il faut encore ajouter une sortie à l'état initial (car λ est dans L*).



Exercice. Expliquer pourquoi, dans cette dernière construction (pour l'étoile de Kleene), il faut éviter de partir d'un λ–automate dont l'état initial est récursif.

TRANSFORMATION DES GRAMMAIRES ALGEBRIQUES

Note. Ce chapitre, plus technique que les précédents, est un complément à la présentation des algorithmes d'analyse syntaxiques. Il est indépendant du chapitre précédent qui, traitant des automates est lié essentiellement aux grammaires régulières.

Le problème : on a vu que l'analyse syntaxique peut être longue, et même inopérante, selon la forme des règles de grammaire. Les règles qui commencent par une variable, et particulièrement celles de la forme $A \rightarrow Au$ (où u est une chaîne quelconque), dites directement récursives à gauche, peuvent empêcher l'analyse syntaxique d'aboutir.

Une solution : décrire un algorithme permettant de modifier la grammaire G donnée de manière à obtenir une grammaire équivalente (c'est-à-dire qui engendre le même langage*) dont toutes les règles non vides débutent par un symbole terminal*. De plus, pour accélérer encore l'analyse, on peut exiger que seul l'axiome peut avoir une règle vide (dans le cas où la chaîne vide appartient au langage $L(G)$).

La transformation consiste en une suite d'opérations successives, qui seront décrites à l'aide d'algorithmes et donc programmables. Résumons-en les étapes consécutives :

1. suppression de la récursivité de l'axiome
2. suppression des règles vides
3. suppression des enchaînements de variables
4. suppression des variables et symboles inutiles
5. mise sous forme normale de Chomsky
6. élimination de la récursivité directe à gauche
7. mise sous forme normale de Greibach

1. Suppression de la récursivité de l'axiome.

Le but est d'éviter un retour à l'axiome au cours des dérivations, autrement dit d'éviter les dérivations de la forme $S \Rightarrow uSv$. Pour cela, l'axiome ne doit pas figurer dans les règles de grammaire (à droite de la flèche, bien entendu).

Si la variable axiome S est récursive, c'est-à-dire s'il existe une règle de la forme $A \rightarrow uSv$:

- rajouter une nouvelle variable, disons S' , qui sera dorénavant l'axiome (la variable S est de ce fait banalisée)
- rajouter (en tête de liste) la règle de grammaire $S' \rightarrow S$

Exemple. G : $S \rightarrow aS \mid A$ (l'axiome est S)
$A \rightarrow b \mid \lambda$

La nouvelle grammaire est

G' : $S' \rightarrow S$ (l'axiome est S')
$S \rightarrow aS \mid A$ (S est une variable banale)
$A \rightarrow b \mid \lambda$

et on a évidemment $L(G) = L(G')$.

2. Suppression des règles vides.

L'opération précédente est supposée déjà faite.

Pour une meilleure lisibilité, nous supposons aussi que les variables ont été renommées de manière à ce que la notation S désigne de nouveau l'axiome.

L'existence de variables desquelles dérive la chaîne vide ralentit l'analyse syntaxique. Il s'agit donc d'éviter les dérivations de la forme $A \Rightarrow \lambda$, dans la mesure du possible : en effet, si la chaîne vide appartient au langage engendré, nous avons nécessairement une dérivation $S \Rightarrow \lambda$, et celle-ci devra être rester. Un algorithme construira récursivement l'ensemble des variables concernées.

De telles dérivations ne peuvent exister que s'il y a des *règles vides*, autrement dit des règles de la forme $A \to \lambda$. On peut supprimer ces règles vides à condition de compenser l'effet de cette suppression par l'ajout de nouvelles règles adéquates.

2.1. Construction de l'ensemble NUL des variables dont dérive la chaîne vide.

Voici un algorithme construisant récursivement l'ensemble NUL.

1. initialiser NUL = { A : variable telle que $A \to \lambda$ soit une règle de grammaire}
2. répéter
   pour chaque règle $A \to w$ telle que $w \in (NUL)^*$ et A non encore dans NUL, faire
   $NUL = NUL \cup \{A\}$
   tant que l'ensemble NUL n'est pas stationnaire (c'est-à-dire jusqu'à ce qu'un passage en revue complet de toutes les règles laisse l'ensemble NUL inchangé).

Notons que $\lambda \in L(G)$ si et seulement si $S \in NUL$ à la fin de la construction.

Exemple. G : $S \to T$ (l'axiome est S)
$T \to aT \mid A$
$A \to b \mid \lambda$

Les états successifs de l'ensemble sont :	NUL = { A }	initialisation
	NUL = { A, T }	car règle $T \to A$
	NUL = { A, T, S }	car règle $S \to T$

2.2. Modification de la grammaire.

1. Pour chaque règle $A \to w$ de la grammaire :
   Repérer dans la chaîne w toutes les occurrences de variables collectées dans NUL.
   Sélectionner un nombre arbitraire de ces occurrences et les supprimer : si la chaîne ainsi obtenue n'est pas vide, ajouter à la grammaire la nouvelle règle ainsi obtenue. Faire ceci pour tous les choix possibles.
   Plus précisément, si la chaîne w contient n occurrences de variables figurant dans NUL , nous avons n! choix différents (y compris aussi zéro suppression) et donc, à priori, n! règles (y compris la règle $A \to w$) ; toutefois, l'une d'elles peut être vide (c'est le cas si $w \in (NUL)^*$) et de plus, ces règles ne sont pas nécessairement distinctes ou nouvelles. On rajoute à la grammaire toutes les règles <u>non vides</u> ainsi obtenues, si elles n'y figurent pas encore.
2. Supprimer toutes les règles vides.
3. Si S (l'axiome) est dans NUL, rajouter la règle $S \to \lambda$.

Exemple. G : $S \to T$
$T \to aT \mid AA$
$A \to b \mid bATA \mid \lambda$

On obtient NUL = { A, T, S }. Comme l'axiome $S \in$NUL, la chaîne vide appartient à L(G) et nous devrons rajouter, « à la main », la règle vide pour l'axiome.
La nouvelle grammaire est

G' : $S \to T \mid \lambda$
$T \to aT \mid a \mid AA \mid A$
$A \to b \mid bATA \mid bTA \mid bAA \mid bAT \mid bA \mid bT$

Exercice. Comment les différents sous-ensembles d'occurrences de variables NUL peuvent-ils être « passés en revue », autrement dit ordonnés (de manière algorithmique) ?

3. Suppression des enchaînements de variables.

Les opérations précédentes sont supposées déjà faites.
Les règles *d'enchaînement de variables* sont de la forme $A \to B$ (où B est une variable). Elles rallongent l'analyse syntaxique en laissant le préfixe terminal inchangé ; on cherche donc à les éliminer. Ces règles pourront être supprimées à condition que cette suppression soit compensée par l'ajout de nouvelles règles adéquates.

On commence par construire, pour chaque variable, l'ensemble des variables qu'on atteint par des règles d'enchaînement.

3.1. Construction de l'ensemble CHAIN(A) pour chaque variable A .

Par définition, CHAIN(A) = { B : variable telle qu'il existe une dérivation A => B }

Voici un algorithme construisant récursivement cet ensemble.

1. initialiser CHAIN(A) = { A }
2. répéter
   s'il y a une règle $B \to C$ avec $B \in$ CHAIN(A) telle que C soit une variable non encore dans CHAIN(A), faire
   CHAIN(A) = CHAIN(A) $\cup$ { C }
   tant que l'ensemble CHAIN(A) n'est pas stationnaire (c'est-à-dire jusqu'à ce qu'un passage en revue complet de toutes les règles laisse l'ensemble inchangé).

3.2. Modification de la grammaire.

1. Pour chaque variable A ,
   pour chaque variable $B \in$ CHAIN(A)
   pour chaque règle $B \to w$ telle que w ne soit pas une variable, rajouter la règle $A \to w$
2. Supprimer toutes les règles d'enchaînement de variables.

Exemple. G : $S \to T \mid \lambda$
$T \to aT \mid AA \mid A$
$A \to b$

Nous obtenons $CHAIN(S) = \{S, T, A\}$
$CHAIN(T) = \{T, A\}$
$CHAIN(A) = \{A\}$

et la nouvelle grammaire est G' : $S \rightarrow \lambda \mid aT \mid AA \mid b$
$T \rightarrow aT \mid AA \mid b$
$A \rightarrow b$

4. Suppression des variables et symboles inutiles.

Les opérations précédentes sont supposées déjà faites.
Un élément (variable ou symbole) $X \in V \cup \Sigma$ est dit *utile* s'il existe une dérivation

$$S \Rightarrow uXv \Rightarrow w \in \Sigma^*$$

En partant de l'axiome, il faut donc arriver à une chaîne terminale en transitant par une chaîne dans laquelle figure X . Le but de cette étape est de « faire le ménage » parmi les variables, après les suppressions de règles faites précédemment.

On commence par collecter dans un ensemble TERM les variables dont dérivent des chaînes terminales, puis on construit le sous-ensemble ATTEINT des variables qu'on peut atteindre à partir de l'axiome. Les variables qui ne sont pas dans ATTEINT seront supprimées, avec leurs règles de grammaire. Par cette opération, des symboles terminaux peuvent aussi avoir disparu, s'ils ne figurent pas dans les règles restantes.

4.1. Construction de l'ensemble des variables dont dérivent des chaînes terminales

Soit l'ensemble des variables dont dérivent des chaînes terminales

$$TERM = \{ A : \text{variable telle qu'il existe une dérivation } A \Rightarrow w \in \Sigma^* \}$$

Il est obtenu récursivement par l'algorithme suivant.

1. initialiser $TERM = \{ A : \text{variable telle qu'il existe une règle } A \rightarrow u \text{ avec } u \in \Sigma^* \}$
2. répéter
   s'il y a une règle $A \rightarrow u$ avec $u \in (\Sigma \cup TERM)^*$ et A non encore dans TERM,
   faire $TERM = TERM \cup \{ A \}$
   tant que l'ensemble TERM n'est pas stationnaire.

4.2. Modification de la grammaire.

Elle est simplissime : on ne garde que les variables collectées dans TERM, ainsi que les règles qui les concernent.

4.3. Construction de l'ensemble des variables qu'on atteint à partir de l'axiome.

L'étape TERM est supposée faite ; on peut donc obtenir une chaîne terminale à partir de chaque variable restante de la grammaire. On définit l'ensemble

$$ATTEINT = \{ A : \text{variable telle qu'il existe une dérivation } S \Rightarrow uAv \}$$

Cet ensemble est construit récursivement de la manière suivante.

1. initialiser ATTEINT = { S }
2. répéter
   pour chaque une règle $A \to u$ avec $A \in$ ATTEINT,
   pour chaque variable B figurant dans u faire ATTEINT = ATTEINT $\cup$ { B }
   tant que l'ensemble ATTEINT n'est pas stationnaire.

4.4. Modification de la grammaire.

Comme auparavant, on ne garde que les variables collectées dans ATTEINT, et les règles qui les concernent.

De ce fait, des symboles terminaux peuvent éventuellement être supprimés : c'est le cas pour ceux qui ne figuraient que dans des règles concernant des variables supprimées.

5. Forme normale de Chomsky (1959).

Définition. Une grammaire G est sous *forme normale de Chomsky* si toutes ses règles sont de l'un des types suivants :

- $A \to BC$ où $B, C \in V-\{S\}$
- $A \to a$ où $a \in \Sigma$
- $S \to \lambda$ où S est l'axiome (cette règle est utilisée lorsque λ est dans le langage).

Théorème 9. Toute grammaire algébrique est équivalente à une grammaire sous forme normale de Chomsky.

Preuve du théorème 9.

Faire, si nécessaire, les transformations précédentes 1, 2, 3 et 4. Il reste à modifier les règles non conformes $A \to w$ où $w \in (\Sigma \cup V)^+$.

Comme les règles d'enchaînements de variables ont déjà été supprimées, la chaîne w est de longueur supérieure ou égale à deux. Tout d'abord, on introduit des nouvelles variables de manière à pouvoir remplacer w par une chaîne de même longueur mais sans symboles terminaux. Ensuite, les règles de longueur strictement supérieure à deux pourront être remplacées par des règles plus courtes grâce à des variables supplémentaires introduites dans ce but.

Exemple.

Soit la règle $A \to aBAdC$ où $a, d \in \Sigma$ et $A, B, C \in V$.

1) On introduit des variables nouvelles pour chacun des symboles terminaux figurant dans la chaîne. La règle précédente est ainsi remplacée par le groupe de règles

$$\begin{cases} A \to A'BAD'C \\ A' \to a \\ D' \to d \end{cases}$$

2) Il reste à modifier la règle de longueur supérieure à deux $A \to A'BAD'C$.

En introduisant encore de nouvelles variables, cette règle peut être remplacée par le groupe de règles suivantes, conformes au modèle de Chomsky

$$\begin{cases} A \to A'A_1 \\ A_1 \to BA_2 \\ A_2 \to AA_3 \\ A_3 \to D'C \end{cases}$$

6. Forme normale de Greibach (1965).

Définition. Une grammaire G est sous *forme normale de Greibach* si toutes ses règles sont de l'un des types suivants :

- $A \to a$ où $a \in \Sigma$
- $A \to a A_1 A_2 \ldots A_n$ où $a \in \Sigma$ et $A_1, A_2, \ldots, A_n \in V-\{S\}$
- $S \to \lambda$ où S est l'axiome (cette règle est utilisée lorsque λ est dans le langage).

L'intérêt de cette forme de grammaire est évident : le préfixe terminal des chaînes est rallongé à chaque utilisation d'une règle de grammaire.

Théorème 10. Toute grammaire algébrique est équivalente à une grammaire sous forme normale de Greibach.

6.1. Le plan de la preuve du théorème 10.

On fait, si nécessaire, les transformations précédentes 1, 2, 3, 4 et 5. A partir d'une grammaire sous forme normale de Chomsky, il reste donc à modifier les règles de la forme $A \to BC$.

L'idée est, bien entendu, de réécrire la première variable avec toutes les règles qui la concernent, et ceci de manière répétée jusqu'à obtenir des règles commençant par un symbole terminal. Malheureusement, on peut entrer ainsi dans une boucle sans fin ; par exemple avec le groupe de règles

$A \to BC$
$B \to AD$

on obtiendra $A \to ADC$ puis $A \to BCDC$ etc… Dans cette situation, il apparaît au cours des réécritures successives des règles dites *directement récursives à gauche* c'est-à-dire de la forme $A \to Au$.

Dans une étape préliminaire un peu technique, nous allons voir comment il est possible de supprimer les règles directement récursives à gauche à l'aide d'une nouvelle variable (qui sera, elle, directement récursive à droite, ce qui ne présente aucun inconvénient).

Après avoir fixé un ordre pour les variables, cette technique nous permettra de progresser vers une forme « presque Greibach » dans laquelle les règles de grammaire pour une variable A donnée commencent soit par un symbole terminal, soit par une variable dont le numéro d'ordre est strictement supérieur à celui de A . Dans cette situation, les réécritures des règles ne peuvent qu'augmenter encore le numéro de la variable en préfixe, et aboutir à un symbole terminal en préfixe, après un nombre fini d'étapes.

6.2. La technique d'élimination de la récursivité directe à gauche.

Définition. Une règle *directement récursive à gauche* est une règle de la forme $A \to Au$ où $u \in (\Sigma \cup V)^+$.

Nous avons vu que ce type de règle est particulièrement gênant pour l'analyse syntaxique, et nous allons décrire une technique permettant de remplacer la récursivité directe à gauche par de la récursivité directe à droite, qui ne présente aucun désavantage particulier.

Considérons l'ensemble des règles pour une variable A donnée. S'il y a une (ou plusieurs) règles directement récursives à gauche, il y a aussi d'autres règles qui ne le sont pas, puisque A n'a pas été éliminée lors de la transformation 4 (suppression des variables inutiles). Une de ces autres règles est nécessairement utilisée à la suite d'une séquence d'applications de règles directement récursives à gauche. En introduisant une nouvelle variable, il est possible d'obtenir (avec une récursivité directe à droite sur cette variable auxiliaire) toutes les chaînes qui dérivent de A par les règles directement récursives à gauche. Un exemple :

$$A \to Au \mid v \mid w$$

où v et w ne commencent pas par A (ce sont les deux règles non récursives à gauche).

En appliquant la première règle n fois de suite ($n \geq 1$), on produit la chaîne Au^n. On poursuit soit avec la deuxième règle, ce qui nous donne $v.u^n$, soit avec la troisième, et nous obtenons ainsi $w.u^n$.

Il faut pouvoir obtenir ces chaînes $v.u^n$ et $w.u^n$ d'une autre manière à partir de A. Pour cela, il nous suffit de remplacer toutes les règles concernant A par le groupe de règles suivant :

$$A \to vZ \mid wZ \mid v \mid w$$
$$Z \to uZ \mid u$$ (Z est une nouvelle variable introduite)

On vérifie que la variable auxiliaire sert à produire toutes les chaînes de la forme u^n, et celles-ci sont ensuite concaténées à u ou à v. Elle est directement récursive à droite.

Ce petit exemple peut facilement se généraliser au cas où il y a un nombre quelconque de règles :

$$A \to Au_1 \mid Au_2 \mid \ldots \mid Au_p \mid v_1 \mid v_2 \mid \ldots \mid v_q$$

avec p règles directement récursives à gauche, et q règles qui ne le sont pas. On obtient les nouvelles règles

$$A \to v_1Z \mid \ldots \mid v_qZ \mid v_1 \mid v_2 \mid \ldots \mid v_q$$
$$Z \to u_1Z \mid \ldots \mid u_pZ \mid u_1 \mid \ldots \mid u_p$$

Exemple. Voici un autre exemple, concret, sous forme normale de Chomsky, avec deux règles directement récursives à gauche :

$$A \to AB \mid AC \mid a \mid BA$$

Les chaînes dérivées de A en appliquant uniquement les règles directement récursives à gauche sont $A.\{B, C\}^+$. Il faut ensuite appliquer avec la troisième ou la quatrième règle, ce qui nous donne $a.\{B, C\}^+$ et $BA.\{B, C\}^+$. On peut donc remplacer les règles pour A par le groupe de règles

$$A \to aZ \mid BAZ \mid a \mid BA$$
$$Z \to BZ \mid CZ \mid B \mid C$$ (Z désigne la nouvelle variable introduite)

On peut remarquer à cette occasion que ces nouvelles règles ne sont plus sous forme de Chomsky : nous obtenons des chaînes de variables avec éventuellement un symbole terminal en préfixe ; de plus, des règles d'enchaînements de variables sont ainsi réapparues, mais uniquement pour la variable auxiliaire qui a été introduite.

6.3. De la forme de Chomsky à la forme « presque Greibach ».

Commençons par choisir un ordre sur les variables, l'axiome étant la première variable.

Définition. Une grammaire dont les variables sont ordonnées est de la forme *presque Greibach* si les règles sont de l'un des types suivants :

- $S \to \lambda$
- $A \to a.u$ où $a \in \Sigma$ et $u \in V^*$
- $A \to Bu$ où $u \in V^+$ et B est une variable telle que $n°(B) > n°(A)$.

Notre but est d'arriver à cette forme, à partir d'une forme de Chomsky.

On procède récursivement, en suivant l'ordre des variables.

On commence donc par l'axiome : $n°(S)=1$. Comme l'axiome est non récursif (cf. transformation 1), ses règles sont déjà de la forme « presque Greibach » voulue.

Supposons que les $i-1$ premières variables ont déjà été traitées et sont sous la forme « presque Greibach ». Considérons les règles pour la $i^{ème}$ variable. On réécrit celles qui commencent par une variable dont le n° est strictement inférieur à i avec les règles déjà obtenues : ceci augmente le n° de la variable figurant en préfixe, ou donne un symbole terminal en préfixe. Cette opération de réécriture est répétée jusqu'à ce qu'aucune règle ne commence plus par une variable dont le n° est strictement inférieur à i .

Pour notre $i^{ème}$ variable, nous avons maintenant des règles qui sont directement récursives à gauche (ce sont celles qui ont la $i^{ème}$ variable en préfixe), ou qui sont déjà sous la forme « presque Greibach » voulue. On supprime la récursivité directe à gauche pour la $i^{ème}$ variable (cf. 6.2) en introduisant une nouvelle variable ad hoc (placée en fin de liste). Toutes les règles pour la $i^{ème}$ variable sont maintenant de la forme souhaitée.

Le processus s'arrête car les nouvelles variables introduites n'auront pas de règle directement récursive à gauche, puisque ces variables ne figurent en préfixe d'aucune règle pour les variables précédentes. On n'ajoute donc pas indéfiniment de nouvelles variables…

6.4. De la forme « presque Greibach » à la forme de Greibach.

Pour achever la preuve du théorème 10, il suffit de partir d'une grammaire sous la forme presque Greibach. Seules les règles qui commencent par une variable doivent encore être modifiées. Leur réécriture avec les règles concernant cette variable en préfixe ne peut qu'augmenter le n° de la variable en préfixe, ou bien donner un symbole terminal en préfixe. Comme il n'y a qu'un nombre fini de variables et de règles, la transformation est achevée après un nombre fini de telles réécritures..

NETOGRAPHIE

Une présentation historique du FORTRAN
http://community.computerhistory.org/scc/projects/FORTRAN/
http://www.ibiblio.org/pub/languages/fortran/ch1-1.html

Une biographie et une présentation historique des travaux de John W. Backus
http://www.thocp.net/biographies/backus_john.htm
http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Backus.html

Une présentation de la notation BNF (BNF, "Backus-Naur Form") pour les grammaires, introduite pour le langage ALGOL58 et mise au point en 1960 par John Backus et Peter Naur :
http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html

Un site pour voir de quoi a l'air une "vraie" grammaire : on y trouve une grammaire du PASCAL
http://www.irietools.com/iriepascal/progref534.html#appendix_h_grammar
et, plus généralement, des informations sur la structure du langage PASCAL :
http://www.irietools.com/iriepascal/progref.html

Pour un aperçu historique plus général (« musée virtuel ») de l'informatique :
http://vmoc.museophile.org/
http://www.fh-jena.de/~kleine/history/

Une biographie du mathématicien S.C. Kleene
http://www.onelang.com/encyclopedia/index.php/Stephen_Kleene
et celle de Noam Chomsky, un personnage présenté dans toute sa diversité et ses évolutions au fil des ans
http://www.onelang.com/encyclopedia/index.php/Noam_Chomsky

Les références bibliographiques suivantes sont celles des publications originelles des résultats fondamentaux qui ont été mentionnés dans le cours.

Introduction par S. Kleene, en 1956, de la notion de langage régulier :

Kleene, S. , *Representation of Events in Nerve Nets and Finite Automata* in Automata Studies (1956) eds. C. Shannon and J. McCarthy.

Les travaux fondamentaux de N. Chomsky sur la description et la classification des langages et grammaires :

1. Chomsky, N., *Three models for the description of language*, IRE Transactions on Information Theory, 2 (1956), pages 113-124
2. Chomsky, N., *On certain formal properties of grammars*, Information and Control, 1 (1959), pages 91-112

L'article de Sheila A. Greibach introduisant, en 1965, la forme normale de Greibach, a été l'une de ses premières publications :

Greibach, S., *A New Normal-Form Theorem for Context-Free Phrase Structure Grammars,* Journal of the ACM (JACM), Volume 12 Issue 1 (1965)