ECOLE SUPERIEURE D'INGENIEURS DE LUMINY

Département Informatique

THEORIE DES LANGAGES

Notes de Cours

Henry Kanoui

Janvier 2005

THEORIE DES LANGAGES

ESIL – DEPARTEMENT INFORMATIQUE – 1ERE ANNEE

Horaires

- Volume total : 12 h de cours + 12 h de TD (2 groupes)

Références bibliographiques

- *Théorie des langages et des automates* par J.M. Autebert, Masson (1994)

- *Introduction to Automata Theory, Languages and Computation* par J.E. Hopcroft, R. Mtwani et J.D. Ullman – 2nd ed. Addison-Wesley (2001)

Résumé

L'origine de la théorie des langages est un essai de modélisation des langues naturelles. Son expansion tient toutefois à son adéquation à la description des langages de programmation. En effet, les notions de langage, de grammaire, de dérivation et d'automates sont à la base de tous les algorithmes d'analyse syntaxique, et donc de l'écriture des compilateurs. Par ailleurs, la théorie des automates finis fournit le moyen de modéliser de nombreux problèmes.

Ce cours débute par des exemples d'application pratique des automates qui conduisent ensuite à préciser le cadre formel de la théorie des langages. On entre dans le vif du sujet par l'étude des automates d'états finis puis des langages réguliers et des grammaires associées et on montre l'équivalence entre ces notions.

La deuxième partie est consacrée aux automates à pile et aux grammaires et langages hors contexte. L'étude des formes normales de ces grammaires permet de conclure sur les équivalences entre ces notions. On termine par le sous ensemble intéressant que sont les langages déterministes et les techniques d'analyse LL(1) associées.

SOMMAIRE

CHAPITRE I : INTRODUCTION : AUTOMATES, LANGAGES ET GRAMMAIRES 4

UN EXEMPLE D'APPLICATION DES AUTOMATES FINIS 6
CONCEPTS PRINCIPAUX DE LA THEORIE DES LANGAGES 10

CHAPITRE II : AUTOMATES D'ETATS FINIS 12

DEFINITIONS ET PROPRIETES FONDAMENTALES 12
AEF DETERMINISTE 13
AEF AVEC ε-TRANSITIONS 17

CHAPITRE III : SYSTEMES DE REECRITURE - GRAMMAIRES 21

SYSTEMES DE REECRITURE 21
GRAMMAIRES 22

CHAPITRE IV : GRAMMAIRES ET LANGAGES REGULIERS 25

EXPRESSION REGULIERE 25
AEF ET LANGAGES REGULIERS 26
GRAMMAIRES REGULIERES ET AEF 28
GRAMMAIRES REGULIERES ET AEF 29
LEMME CARACTERISTIQUE 31

CHAPITRE V : AUTOMATES A PILE 33

DEFINITIONS 33
RECONNAISSANCE/ACCEPTATION 34
EQUIVALENCE MODE PILE VIDE-MODE ETAT FINAL 35

CHAPITRE VI : GRAMMAIRES ET LANGAGES HORS-CONTEXTE 37

EXEMPLES INFORMELS 37
RAPPELS ET PREMIERES DEFINITIONS 38
ARBRES DE DERIVATION - AMBIGUITE 39
FORMES NORMALES DES GRAMMAIRES H.C. 42
LEMME CARACTERISTIQUE 46
EQUIVALENCE AAP-GHC 48

CHAPITRE VII : ANALYSE DESCENDANTE DETERMINISTE – GRAMMAIRES ET LANGAGES LL(1) 52

DEFINITIONS 52
CONDITIONS LL(1) 53
PROPRIETE FONDAMENTALE 53
CALCUL DES ENSEMBLES DIRECTEURS 54
RESULTATS 55

EXERCICES 56

EXERCICES – CHAPITRE I : AUTOMATES ET LANGAGES 56
EXERCICES – CHAPITRE II : AUTOMATES D'ETATS FINIS 57
EXERCICES – CHAPITRE III : SYSTEMES DE RE-ECRITURE ET GRAMMAIRES 58
EXERCICES – CHAPITRE IV : GRAMMAIRES ET LANGAGES REGULIERS 59
EXERCICES – CHAPITRE V : AUTOMATES A PILE 60
EXERCICES – CHAPITRE VI : GRAMMAIRES HORS-CONTEXTE 61
EXERCICES – CHAPITRE VII : LANGAGES LL(1) 62

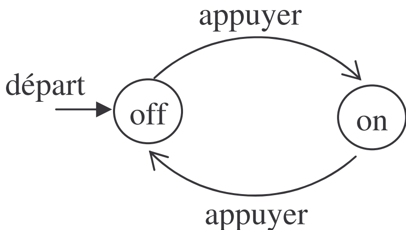
Chapitre I : INTRODUCTION : AUTOMATES, LANGAGES ET GRAMMAIRES

Les ***automates finis*** procurent un modèle pour une large catégorie de programmes et de machines, par exemple :

- conception et test de circuits intégrés
- analyse lexicale pour la compilation
- scanning de textes volumineux (pages web) et recherche d'occurrences de mots, de locutions ou d'autres formes alphanumériques
- modélisation/validation de systèmes de toutes sortes ayant un nombre fini d'états distincts, comme les protocoles de communication, ou les protocoles d'échange d'information sécurisés.

et en général tout dispositif qui est à chaque instant dans un état donné (parmi un nombre fini).
Le rôle d'un « état » est de mémoriser une partie de l'historique du système considéré. Comme ce nombre d'états est fini, on ne sait pas mémoriser tout l'historique, et il faut donc déterminer ce qu'il est important de retenir et ce qu'on peut oublier.

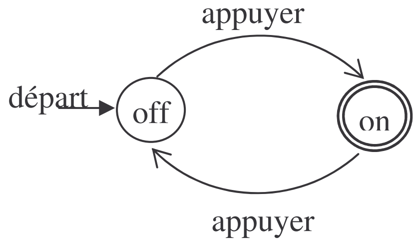
Exemple 1 : le « switch »



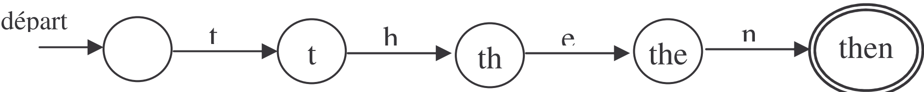
Les états sont représentés par des cercles et sont nommés : ici *on* et *off*. Les arcs orientés faisant passer d'un état à un autre sont étiquetés par les entrées qui représentent des influences (actions) externes sur le système. Ici les 2 arcs sont étiquetés par l'action *appuyer* qui représente un utilisateur appuyant sur le bouton. Quel que soit l'état dans lequel il se trouve, le système change d'état dès qu'il reçoit le signal *appuyer* .

L'un des états est l'état initial, signalé par une flèche. Par convention on a décidé ici que cet état initial est *off*.

Il est souvent nécessaire d'indiquer qu'un état (ou plusieurs) est un état *final*. Se retrouver dans l'un de ces états après une séquence d'entrées indique que cette séquence est « correcte ». Par exemple l'état *on* pourrait être considéré comme un état final car alors le dispositif contrôlé par le switch est en opération. Par convention, un état final est représenté par un cercle double :



Exemple 2
Parfois, un état doit mémoriser quelque chose de plus complexe qu'un choix binaire. Voici un autre automate fini qui pourrait faire partie d'un analyseur lexical et dont le « travail » est de reconnaître le mot clé « then » dans un langage comme Pascal. Il a besoin de 5 états, chacun représentant la position atteinte dans le mot « then » depuis la position initiale (on n'a encore rien lu) jusqu'au mot complet :



Les entrées correspondent aux lettres. On peut imaginer que l'analyseur examine un par un les caractères du programme en cours de compilation et que le caractère à venir est l'entrée de l'automate. L'état initial correspond à la chaîne vide (on n'a encore rien lu) et chaque état a une transition vers la lettre suivante du mot « then ». L'état nommé *then* est atteint quand l'entrée a épelé le mot « then ». Comme le rôle de cet automate est de reconnaître la lecture du mot « then » l'état *then* est le seul état final.

La vision globale que ces deux exemples donnent de l'automate est celle d'une ***machine***. Une machine est décrite par son état, et par un ensemble d'actions ou instructions entraînant une modification de cet état à travers les transitions de l'automate. A cette machine, on associe un ***langage*** qui est l'ensemble de toutes les ***suites*** possibles d'***actions qui ont un sens***. Pour définir ces langages, il faut disposer de formalismes adaptés. Deux de ces formalismes jouent un rôle important dans l'étude des automates et de leurs applications :

1. Les ***grammaires*** sont des modèles utiles pour la description d'objets à structure récursive. Un exemple est donné par l'analyseur, composant d'un compilateur qui traite les structures récursivement imbriquées des langages de programmation (expressions arithmétiques, conditionnelles, etc.). Par exemple une règle de grammaire comme : $E \rightarrow E + E$ exprime qu'une expression arithmétique peut être construite à partir de 2 autres expressions arithmétiques avec un signe '+' entre elles.

2. Les ***expressions régulières*** servent elles aussi à décrire la structure des données, et en particulier les chaînes de caractères. Nous verrons qu'elles décrivent exactement la même chose que les automates finis. Par exemple, l'expression régulière de style UNIX :

   ```
   '[A-Z][a-z]*[ ][0-9][0-9]'
   ```

   représente un mot dont l'initiale est une lettre capitale suivi d'un blanc et de 2 chiffres comme « Paris 75 », mais pas « Aix-en-Provence 13 » qui nécessiterait une expression plus complexe :

   ```
   '[A-Z][a-z]*{-[A-Z,a-z][a-z]*}*[ ][0-9][0-9]'
   ```

   Dans cette notation :
   - [A-Z] représente l'ensemble des caractères entre A et Z majuscules
   - [ ] représente le caractère « espace »
   - « * » signifie « répéter un nombre arbitraire de fois » l'expression qui précède
   - les accolades sont des délimiteurs qui ne font pas partie du texte décrit par l'expression.

L'objet de ce cours est justement de faire la liaison entre d'une part les langages, et d'autre part les grammaires (description) et les automates (exécution).

UN EXEMPLE D'APPLICATION DES AUTOMATES FINIS

Ce qui suit est l'exemple d'un problème réel dont la solution utilise les automates finis. On s'intéresse à des protocoles de paiement électronique qu'un consommateur peut utiliser pour acheter et payer des biens sur l'internet, le vendeur étant assuré que cet argent existe réellement. Pour cela le vendeur doit savoir que l'« ordre de paiement électronique » matérialisé par un fichier informatique n'a pas été fabriqué pour l'occasion, ni copié et envoyé au vendeur alors que l'acheteur garde l'original pour d'autres achats.

L'authenticité du fichier « ordre de paiement » est assuré par une banque et une politique d'encryptage. Donc une tierce partie, la banque, doit émettre et encrypter le fichier « ordre de paiement ». La banque a une autre responsabilité qui est de garder la trace de toutes les opérations.

Règles du jeu

Il y a 3 protagonistes : le client, le magasin et la banque. Pour simplifier, on suppose qu'il n'y a qu'un seul compte. Voici les événements possibles :

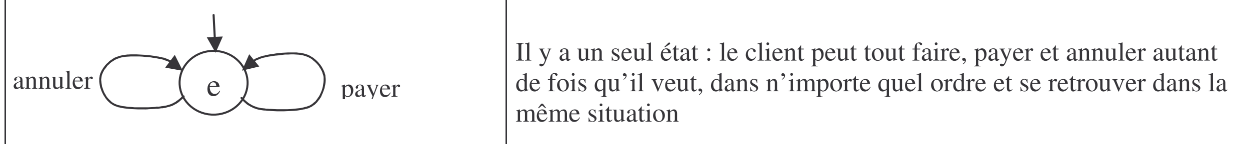
1. Le client décide de *payer*. C'est à dire qu'il demande à la banque d'envoyer un ordre de paiement au magasin
2. Le client décide d'*annuler*. Un ordre de paiement est envoyé à la banque accompagné d'un message demandant que le montant soit reversé sur le compte du client.
3. Le magasin *livre* les marchandises au client
4. Le magasin *endosse* l'ordre de paiement. C'est à dire que l'ordre est renvoyé à la banque avec demande de créditer le compte du magasin.
5. La banque exécute l'ordre de paiement, en créant un nouveau fichier encrypté et *crédite* le compte du magasin.

Le protocole

Les 3 protagonistes doivent soigneusement définir leurs comportements respectifs pour que le fonctionnement du système complet reste sûr et cohérent. Par exemple le client pourrait essayer de reproduire l'ordre de paiement (c'est un fichier), l'utiliser pour payer plusieurs choses, ou payer puis annuler sa commande pour avoir les marchandises gratuitement.
La banque doit assurer que 2 magasins n'endossent pas le même paiement et ne doit pas permettre qu'un ordre de paiement soit à la fois annulé et endossé.
Le magasin ne doit pas expédier une marchandise avant d'être sûr qu'il a reçu un paiement valide pour cela.

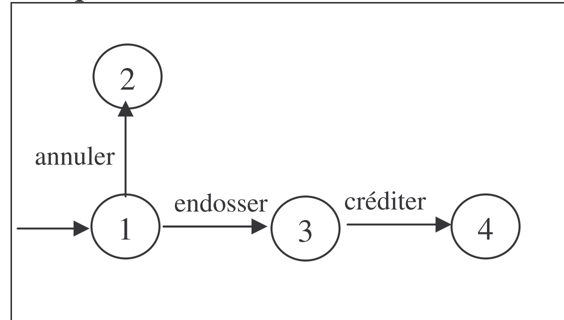
Ce type de protocoles peut être représenté par des automates finis. Chaque état représente une situation dans laquelle peut se trouver un des participants. Ainsi, cet état «mémorise» le fait que certains événements se sont déjà produits et d'autres pas encore. Les transitions entre états ont lieu quand l'un des 5 événements ci-dessus se produit. Ce sont des événements « externes » aux automates représentant les 3 participants, bien que chacun soit responsable de l'un ou l'autre de ces événements. En fait, c'est la succession des événements qui est importante et non qui peut les déclencher. Voici les automates associés à chaque participant :

Le client



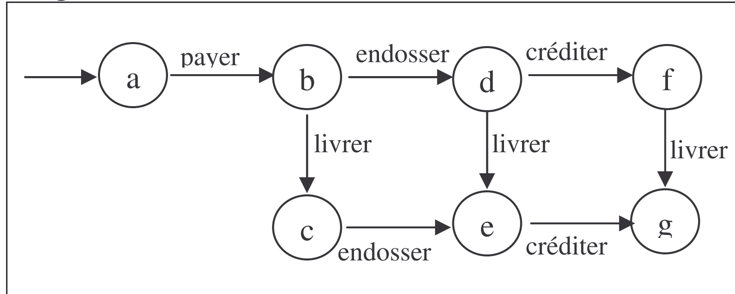
Il y a un seul état : le client peut tout faire, payer et annuler autant de fois qu'il veut, dans n'importe quel ordre et se retrouver dans la même situation

La banque



La situation initiale est celle où la banque a émis l'ordre de paiement mais n'a pas reçu de demande d'endossement ou d'annulation. En cas d'annulation, la banque recrédite le compte du client et passe à l'état 2. Une fois entré dans cet état on y reste : la banque ne permettra pas que le même ordre de paiement soit à nouveau annulé ou dépensé par le client.
Si, dans l'état 1, la banque reçoit un ordre d'endossement elle passe à l'état 3, crédite le compte du magasin et se trouve dans l'état 4 où elle n'acceptera plus d'autre ordre.

Le magasin

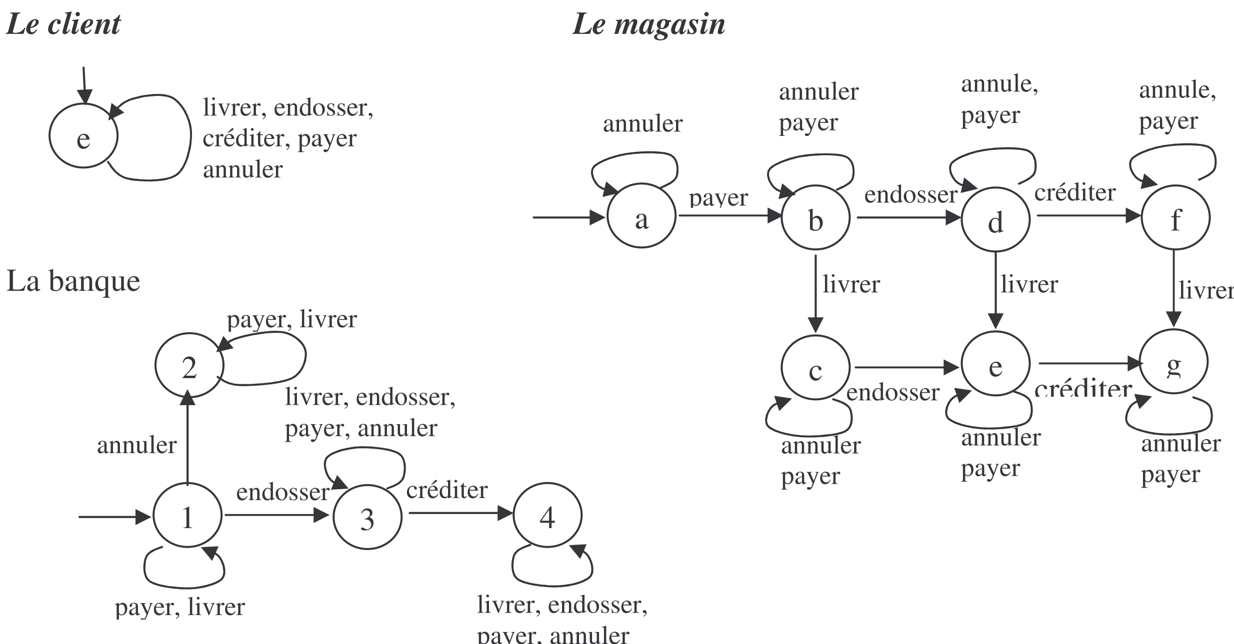


Le départ est en *a*. Dès que le client a donné l'ordre de paiement, on passe en *b* où on peut lancer les actions d'expédition et d'endossement. Il y a quelques défauts : si l'envoi et la facturation sont effectués séparément, alors on peut expédier avant, pendant ou après l'endossement. Le magasin peut se trouver dans une situation où il a expédié la marchandise avant de découvrir que le paiement pose problème.

Ces 3 automates reflètent les comportements des 3 participants, pris indépendamment les uns des autres. Mais il manque certaines transitions : par exemple le magasin n'est pas concerné par l'action *annuler* (du client) alors qu'en ce cas il devrait rester dans l'état où il est. Il faut donc ajouter un arc étiqueté *annuler* de chaque état vers lui même.

Par ailleurs, un des participants peut émettre un message inattendu, par exemple le client peut demander à payer une deuxième fois alors que le magasin est dans l'état *e*. Si c'est le cas, l'automate du magasin s'arrêtera avant d'avoir reçu le transfert de la banque.

En résumé, il faut ajouter à certains états de ces automates des « boucles » pour les actions à ignorer. Voici le résultat.



Un automate pour le système complet

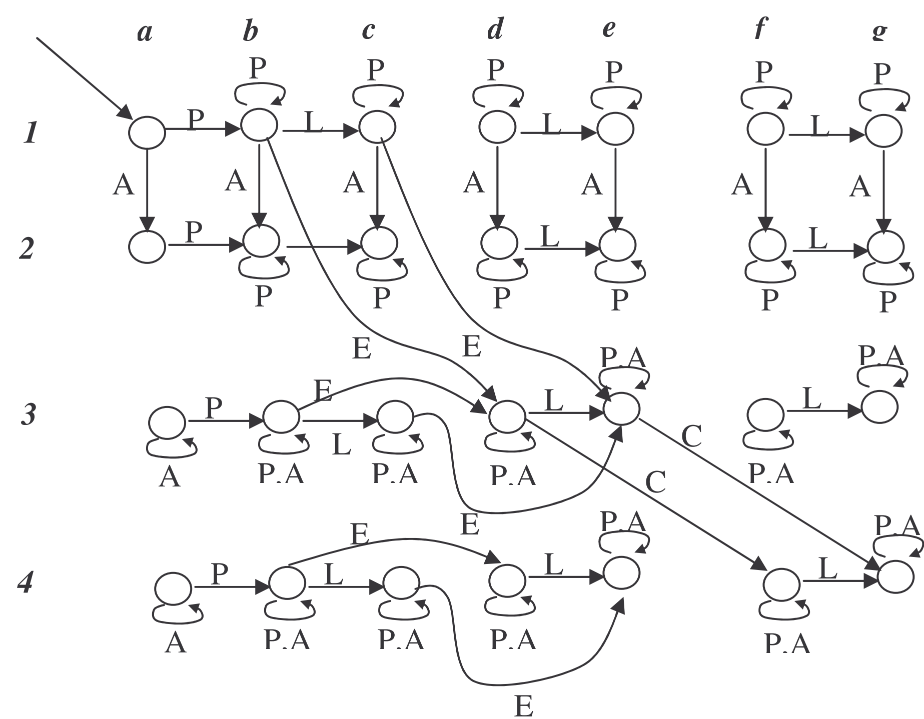
Les 3 modèles individuels des participants ne nous permettent pas de représenter les interactions entre eux. Comme on l'a dit, le client n'a pas de contrainte de comportement et reste toujours dans le même état. Donc le système dans son ensemble ne peut pas s'arrêter si l'automate du client ne répond pas à une sollicitation. D'autre part, la banque et le magasin ont des comportements complexes et il n'est pas évident de déterminer les combinaisons intéressantes de leurs états respectifs.

Le moyen habituel de mettre en évidence les interactions entre ces automates est de construire l'*automate produit*. Chaque état de cet automate est un doublet d'états, un de la banque et un du magasin. Par exemple, l'état (3,*d*) représente la situation où la banque est dans l'état 3 et le magasin dans l'état *d*. Comme la banque a 4 états et le magasin 7, l'automate produit en possède 28.

L'automate produit est donné ci-dessous. Les 28 états sont disposés en tableau : les lignes pour les états de la banque et les colonnes pour ceux du magasin. les actions *payer, livrer, annuler, endosser* et *créditer* ont été abrégées respectivement en *P, L, A, E* et *C.*

Pour construire les arcs, il faut faire tourner les 2 automates (banque et magasin) en parallèle. Chacun d'eux effectue des transitions en fonction des entrées reçues. Si pour une action en entrée donnée, l'un des 2 automates n'a pas de transition prévue, alors l'automate produit s'arrête.

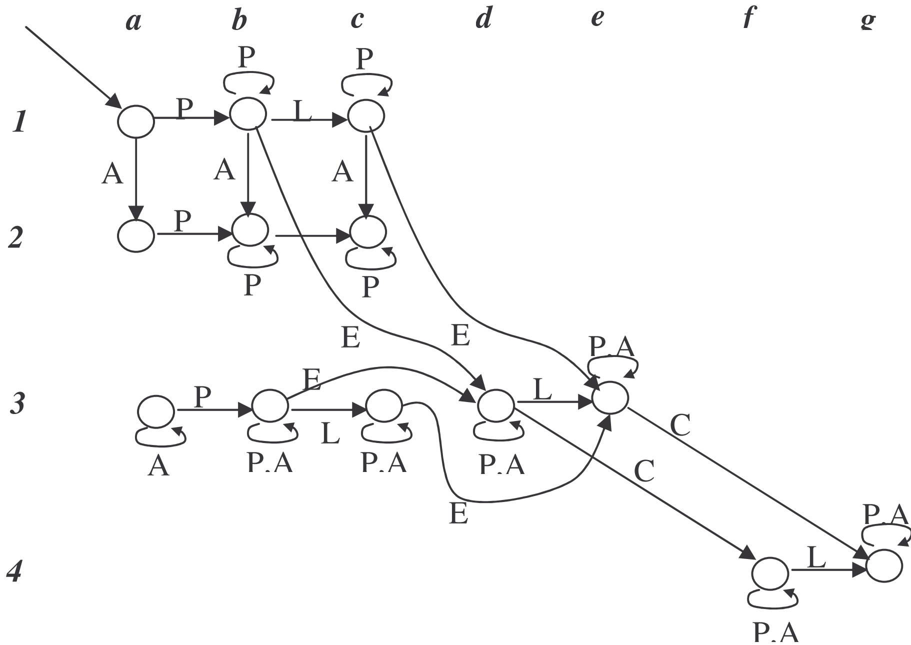
Pour préciser les choses, supposons que l'automate produit soit dans l'état (*i,x*), situation où la banque est dans l'état *i* et le magasin dans l'état *x*. Soit *Z* une action. On cherche si l'automate de la banque a une transition étiquetée *Z* de l'état *i* à l'état *j* et si le magasin a une transition étiquetée *Z* de l'état *x* à l'état *y*. Si *j* et *y* existent tous deux, alors l'automate produit a une transition étiquetée *Z* de l'état (*i,x*), à l'état (*j,y*).



Pour illustrer cela, considérons l'entrée *endosser*. Si la banque reçoit ce message dans l'état 1, elle passe en 3. Dans l'état 3 ou 4 elle reste où elle est et dans l'état 2 l'automate "banque" s'arrête. Pour cette même entrée, le magasin passe de *b* à *d* ou de *c* à *e*. Il y a donc 6 arcs étiquetés *E*. Par exemple dans l'état (1,*b*) l'arc *E* fait passer dans l'état (3,*d*) puisque la banque passe de 1 à 3 et le magasin de *b* à *d*.

Utilisation

Sur les 28 états 10 sont accessibles depuis l'état initial (1,*a*). Par exemple les états (2,*e*) et (4,*d*) ne le sont pas. On peut les éliminer et ne garder que la partie utile de l'automate :



L'intérêt de cet automate vis à vis du protocole qu'il modélise est qu'il permet de répondre à des questions du genre « est ce qu'il est possible que le magasin livre des marchandises qui ne seront pas payées ? », c'est à dire « l'automate produit peut-il entrer dans un état où le magasin a livré (colonne *c, e* ou *g*) et tel que aucune transition *C* n'a été faite ni ne sera faite ? »

Par exemple en (3,*e*) les marchandises ont été livrées mais il y aura de façon certaine une transition *C* vers (4,*g*). Si on regarde la banque, une fois dans l'état 3, elle a reçu la demande d'endossement et l'a traitée. Donc elle est passée par l'état 1 avant l'endossement et donc aucune requête d'annulation n'a été reçue et sera ignorée dans le futur. Donc la banque créditera bien le compte du magasin.

Par contre, l'état (2,*c*) montre qu'il y a un problème : l'état est accessible, mais le seul arc de sortie ramène au même état. C'est la situation où la banque a reçu une demande d'annulation avant l'endossement. Mais le magasin a reçu l'avis de paiement. Le client s'est cru malin et a tout à la fois dépensé et annulé le même argent ! Bêtement, le magasin a livré avant d'endosser le paiement et quand il le fait, la banque n'accusera même pas réception car elle est dans l'état 2 dans lequel l'achat a été annulé.

CONCEPTS PRINCIPAUX DE LA THEORIE DES LANGAGES

Alphabets et mots

Un *alphabet* ou *vocabulaire* est un ensemble fini de symboles appelés lettres. Une *chaîne* ou *mot* sur un alphabet X est une suite finie de lettres de X.
Si p désigne un entier positif, un mot u est une application de l'ensemble {1,2, …,p}dans X ou encore : u : {1,2, …,p}→ X.

L'entier p sera appelé *longueur* du mot et noté |u|. Un *sous-mot* d'un mot u est un mot constitué d'une sous-suite de la suite des lettres de u.
On étend cette définition pour p=0 ce qui introduit le *mot vide* sur X noté 1_X ou plus simplement $\varepsilon : \emptyset \rightarrow X$.

Dans l'ensemble de tous les mots sur un alphabet X on définit l'opération de *concaténation* qui est une loi de composition interne comme suit :
Si u et v sont 2 mots de longueur respective p et q la concaténation de u et v dans cet ordre est le mot w de longueur p+q défini par :
- $w(i) = u(i)$ si $1 \le i \le p$
- $w(p+i) = v(i)$ si $1 \le i \le q$

Cette loi est asssociative et admet ε comme élément neutre. Cette propriété permet de noter un mot par simple juxtaposition de ses lettres.

Puissances d'un alphabet

Si X est un alphabet, on désigne par X^k l'ensemble des mots de X de longueur k. En particulier :
- $X^0 = \{\varepsilon\}$ et ceci quel que soit X.
- X^1 , ensemble des mots de longueur 1 est isomorphe à X. On pourra les confondre.

On désigne par X* (étoile de X) l'ensemble de tous les mots formés sur l'alphabet X, et par X^+ (étoile propre de X) l'ensemble de tous les mots non vides formés sur X, c'est à dire :
- $X^+ = X^1 \cup X^2 \cup X^3 \cup \ldots$
- $X^* = X^0 \cup X^1 \cup X^2 \cup X^3 \cup \ldots = X^+ \cup \{\varepsilon\}$

Langages

On appelle *langage* sur un alphabet X toute partie de X*. Dans l'ensemble P(X*) des parties de X* on définit alors les opérations suivantes :

- opérations booléennes :
  - *union* $(A \cup B)$,
  - *intersection* $(A \cap B)$
  - *complément* à une partie $(A - B)$
- Le *produit* induit par la concaténation : $A.B = \{uv \mid u \in A \text{ et } v \in B\}$
- L'*étoile* d'une partie A notée A* et définie comme suit :
  - $A^0 = \{\varepsilon\}$ et $A^1 = A$ par définition
  - $A^{n+1} = A^n.A$ pour tout entier n
  - $A^* = \bigcup_{n \ge 0} A^n$
- L'*étoile propre* d'une partie A notée $A^+ = \bigcup_{n > 0} A^n$

Propriétés

Le produit est une opération associative et distributive par rapport à l'union

- Le produit n'est pas distributif par rapport à l'intersection
- Si A et B sont des parties d'un alphabet X on a :
  - $(A^*\cup B^*)^* = (A^*.B^*)^* = (A\cup B)^*$
  - $(B.A)^*.B = B.(A.B)^*$
- $\varnothing$, le langage vide, est un langage sur tout alphabet X
- $\{\varepsilon\}$, langage réduit à la chaîne vide, est lui aussi un langage sur tout alphabet X. Il est un élément neutre pour le produit de langages.
- $\{\varepsilon\} \neq \varnothing$: le premier ne contient aucun mot, le second en contient exactement un.

Problèmes

Dans la théorie des automates, un « problème » consiste à décider si une chaîne donnée appartient ou non à un langage donné. Plus précisément, si X est un alphabet et L un langage sur X, alors le « problème de L » est le suivant :

étant donné $w \in X^*$, est ce que oui ou non $w \in L$

Chapitre II : AUTOMATES D'ETATS FINIS

Ce chapitre introduit un schéma générique de description des langages : les automates qui sont des spécifications finies de langages infinis. Nous nous attarderons sur le plus simple d'entre eux : les automates d'états finis (ou AEF). Nous définissons d'abord un AEF en tant que système formel. Dans les chapitres suivants, nous en donnerons une interprétation plus concrète par l'introduction des expressions régulières.

DEFINITIONS ET PROPRIETES FONDAMENTALES

Définition

Un AEF est un 5-uple $<V,Q,D,F,\delta>$ où :

- V est un vocabulaire ou alphabet d'entrée
- Q est un ensemble fini d'états appelés états de l'automate
- $D \subset Q$ est l'ensemble des états initiaux
- $F \subset Q$ est l'ensemble des états finaux
- $\delta \subset Q \times V \times Q$ est l'ensemble des transitions de l'automate

Propriété

La relation δ est étendue à $\delta^* \subset Q \times V^* \times Q$ (et opère donc sur des chaînes et plus seulement sur des symboles) comme suit :

- $(q,\varepsilon,q) \in \delta^*$ pour $q \in Q$
- $(q,a,p) \in \delta^*$ si $(q,a,p) \in \delta$ pour $p,q \in Q$ et $a \in V$
- $(q,ax,p) \in \delta^*$ si $\exists\, r \in Q$ tel que $(q,a,r) \in \delta^*$ et $(r,x,p) \in \delta^*$ pour $q \in Q$, $a \in V$ et $x \in V^*$

On voit que le calcul de δ^* consomme un à un les symboles qui forment la chaîne x jusqu'à ce que celle-ci soit épuisée.

En d'autres termes, si $a_1a_2 \ldots a_n \in V^*$; $p,q \in Q$ on a : $(q, a_1a_2 \ldots a_n , p) \in \delta^*$ ssi il existe une suite d'états $q_0, q_1 \ldots q_n \in Q$ tels que : $q = q_0$, $p = q_n$ et $(q_{i-1}, a_i, q_i) \in \delta$ pour tout $i \in \{1,2,\ldots,n\}$.

L'interprétation de $(q,x,p) \in \delta^*$ est que l'AEF démarre dans l'état $q = q_0$ avec la chaine $x = a_1a_2 \ldots a_n$ en entrée se retrouvera dans l'état $p = q_n$ lorsque la chaine x aura été épuisée.

La suite $(q_0, a_1, q_1)(q_1, a_2, q_2) \ldots (q_{n-1}, a_n, q_n)$ est appelée *chemin* pour $x = a_1a_2 \ldots a_n$
Ce qu'on peut représenter par :

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots\cdots\cdots \; q_{n-1} \xrightarrow{a_n} q_n$$

Désormais et pour simplifier l'écriture, on écrira δ au lieu de δ^*

Représentation sagittale d'un AEF

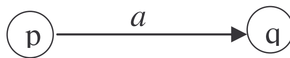
On représente un AEF $A = <V,Q,D,F,\delta>$ par un (multi-)graphe dont :

- les sommets sont les états (éléments de Q).
  un état initial q0 se représente par : → (q0)

  un état final q est représenté par : ((q))

  un état quelconque q est représenté par : (q)

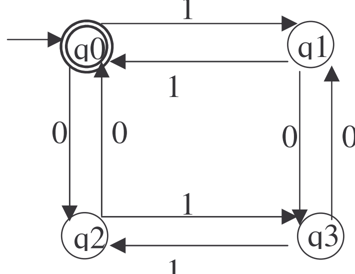
- les arcs du graphes sont étiquetés et associés à la relation de transition :

chaque fois que $\delta(p,a) = q$ pour $p, q \in$ Q et $a \in$ V on a : 

Exemples

A1 : V = {a , b} Q = {1 , 2} D = {1} F = {1 , 2}
δ = {(1,a,1) ; (1,b,2) ; (2,b,2)}

A2 : V = {0,1} Q = {q0,q1,q2,q3}
D = {q0} F = {q0}
δ = { (q0,0,q2) ; (q0,1,q1) ;
(q1,0, q3) ; (q1,1 ;q0) ;
(q2,0,q0) ; (q2,1,q3) ;
(q3,0,q1) ; (q3,1,q2) }



Langage reconnu par un AEF

On dit qu'un mot u $\in$ V* est *accepté* (ou *reconnu*) par un AEF si *q0* $\in$ D est un état initial et s'il existe un état final $q \in$ F tel que $(q0,u, q) \in \delta$

Le *langage accepté* (ou *reconnu*) par un AEF A = <V,Q,D,F,δ> est l'ensemble des mots acceptés (ou reconnus) : L(A) = { $u \in$ V* | $\exists q0 \in$ D et $q \in$ F tel que $(q0,u,q) \in \delta$}

Par exemple, pour les 2 automates précédents :
L(A1) = {chaînes construites sur {a,b} commençant par des « a » et terminées par des « b »}
L(A2) = {chaines construites sur {0,1} ayant un nombre pair de 0 et un nombre pair de 1}

AEF DETERMINISTE

Définition

Un AEF A = <V,Q,D,F,δ> est dit *déterministe* si :
- il n'y a qu'un seul état initial : Card(D) = 1
- une seule transition est possible pour un état et un symbole donné. C'est à dire étant donné q $\in$ Q et a $\in$ V il existe <u>au plus un</u> p $\in$ Q tel que $(q,a,p) \in \delta$.

Dans ce cas on parlera de *fonction de transition* et on écrira $\delta(q,a) = p$ au lieu de $(q,a,p) \in \delta$.
A est dit *déterministe complet* si de plus une transition est toujours possible. C'est à dire étant donné q $\in$ Q et a $\in$ V il existe <u>exactement un</u> p $\in$ Q tel que $(q,a,p) \in \delta$.

Remarque

Tout AEF déterministe non complet peut être complété en un automate déterministe complet. Il suffit d'ajouter un nouvel état q_ω qui ne sera pas un état final et de compléter δ par :
- $\delta(p,a) = q_\omega$ chaque fois que $\delta(p,a)$ n'est pas défini pour q $\in$ Q et a $\in$ V
- $\delta(q_\omega,a) = q_\omega$ pour tout a $\in$ V

Un état tel que q_ω est un état »puits » : dès lors qu'on y arrive, on ne peut plus en sortir !

Théorème
Si A est un AEF, alors il existe un AEF **déterministe** A' tel que L(A') = L(A)

Preuve Soit A= $< V, Q, D, F, \delta >$ et A' = $< V, Q', D', F', \beta >$ l'AEF définit comme suit. Nous allons montrer que A' est déterministe et que L(A') = L(A).

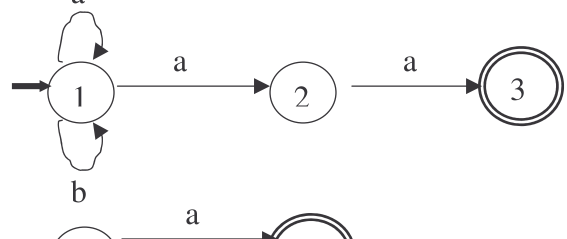
- $Q' = P(Q) - \{\emptyset\}$: ensemble des sous-ensembles non vides de Q. Q' est fini car Q est fini.
- $D' = \{D\}$: un seul état initial qui est la réunion des états initiaux de A
- $\beta: Q' \times V \rightarrow Q'$: définie par $\beta(\{ q_1, q_2,\ldots q_k\},a) = \cup\ \delta (q_j, a)$
- $F' = \{U \in Q' \mid U \cap F \neq \emptyset\}$: sous-ensemble de Q contenant 1 état final de A.

Par construction, A' est un automate déterministe : 1 seul état initial et β est une fonction.

D'autre part : $u \in L(A)$
$\Leftrightarrow$ il y a un chemin w dans A menant de $q_d \in D$ à $q_f \in F$ tq t(w) = u
$\Leftrightarrow \exists\ n > 0\ q_d \in D\ q_f \in F\ q_1, \ldots, q_{n-1} \in Q$ tq $w = (q_d\ x_1\ q_1)\ldots(q_{n-1}\ x_n\ q_f)$ et $u=x_1\ldots x_n$ ou bien : $q_d=q_f$ et $u = \varepsilon$
$\Leftrightarrow \exists\ n > 0\ p_1\ldots p_n \in Q'$ tq $(D\ x_1\ p_1) \ldots (p_{n-1}\ x_n\ p_n)$ est un chemin dans A' pour $u = x_1\ldots x_n$

En effet :
- $q_d \in D \Rightarrow q_1 = \delta (q_d , x_1) \in \beta (D , x_1)$ et ainsi de suite.
- ou bien $D \cap F = \emptyset$ et $u = \varepsilon : \exists\ q \in D \cap F \Rightarrow (D\ \varepsilon\ D)$ est un chemin car :
  $\beta (D , \varepsilon)$ contient $q \Rightarrow \beta(D , \varepsilon) \cap F \neq \emptyset$

A' reconnaît donc le même langage que A.

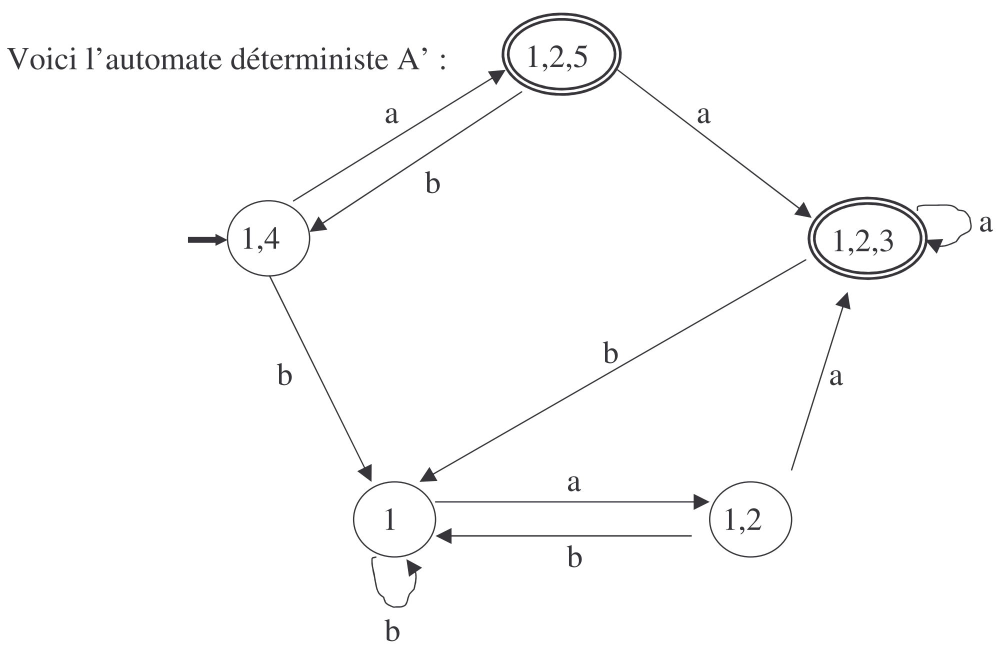


Exemple

Soit l'AEF non déterministeA :
- Etats initiaux : 1 et 4
- Etats finaux : 3 et 5

	a	b	Etapes de la construction de A'
1	1,2	1	
2	3	-	1) reporter les transitions de
3	-	-	l'automate non déterministe
4	5	-	
5	-	4	
1,4	1,2,3	1	2) créer le nouvel état initial
1,2	1,2,3	1	2) saturation au fur et à mesure de la
1,4	1,2,5	1	création de nouveaux états
1,2,3	1,2,3	1	
1	1,2	1	
~~2~~	~~3~~	-	3) simplifier en enlevant :
~~3~~	-	-	- les états d'où rien de part
~~4~~	~~5~~	-	- ceux qui arrivent où il ne part
~~5~~	-	~~4~~	rien
1,2	1,2,3	1	- ceux aux rien n'arrive
1,4	1,2,5	1	et recommencer jusqu'à la stabilité
1,2,3*	1,2,3	1	
1,2,5*	1,2,3	1,4	4) marquer les états finaux (*)

Voici l'automate déterministe A' :



Application : moteur de recherche

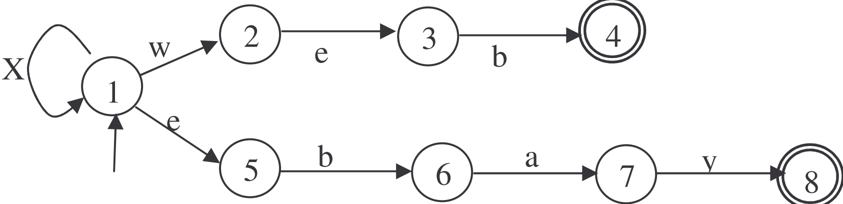
Un problème classique dès qu'on s'intéresse à la gestion de grandes bases de données documentaires (pages web en particulier) est de retrouver tous les documents qui contiennent un ou plusieurs mots donnés. Une des techniques utilisées consiste à gérer un index inversé de tous les mots présents dans la base (de l'ordre de 10^8). Mais cette technique marche mal lorsque la base documentaire évolue trop rapidement ou n'est pas suffisamment formatée.

Une autre technique consiste à utiliser des AEF. Dans ce cas, les textes pertinents sont calculés « à la volée ».

Supposons donné un ensemble de mots appelés *mots-clés* dont on veut retrouver toutes les occurrences. Nous allons définir un AEF qui signale, en état final, qu'un de ces mots a été retrouvé. Cet automate consomme les textes caractère par caractère :

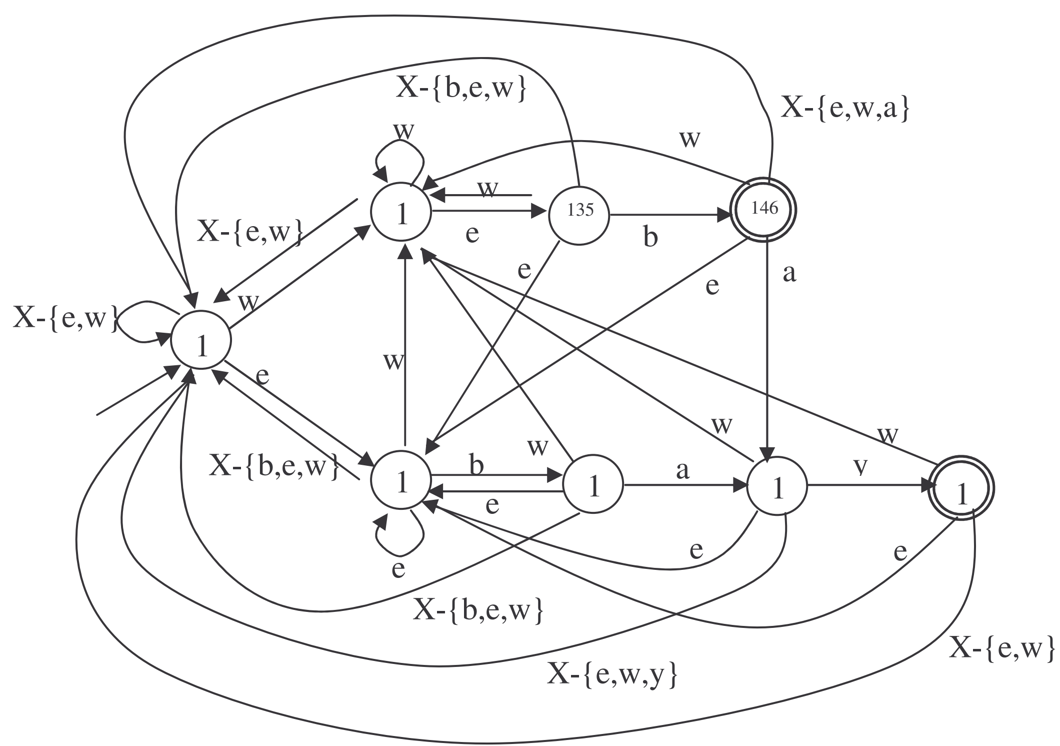
- l'état initial est muni d'une transition sur lui-même pour chaque caractère d'entrée. Intuitivement, il correspond au cas où aucun mot clé n'a été trouvé, même si on en a reconnu quelques lettres du début.
- à chaque mot clé $a_1a_2…a_k$ sont associés k états $q_1, q_2…q_k$. Il y a une transition étiquetée a_1 de l'état initial vers q_1 , une transition étiquetée a_i de l'état q_{i-1} vers q_i et q_k est un état final qui indique que le mot clé a été trouvé.

Voici ce que cela donne pour les mots clés *web* et *ebay* (X représente tout l'alphabet):



Cet AEF est non déterministe et donc très inefficace. On construit l'AEF déterministe équivalent :

	w	e	b	a	y	X-{w,e,b,a,y}
1	12	15	1	1	1	1
~~2~~	-	~~3~~	-	-	-	-
~~3~~	-	-	~~4~~	-	-	-
~~4 (*)~~	-	-	-	-	-	-
~~5~~	-	-	~~6~~	-	-	-
~~6~~	-	-	-	~~7~~	-	-
~~7~~	-	-	-	-	~~8~~	-
~~8 (*)~~	-	-	-	-	-	-
12	12	135	1	1	1	1
15	12	15	16	1	1	1
135	12	15	146	1	1	1
~~14~~	~~12~~	~~15~~	~~1~~	~~1~~	~~1~~	~~1~~
16	12	15	1	1	1	1
17	12	15	1	1	18	1
18 (*)	12	15	1	1	1	1
146	12	15	1	17	1	1

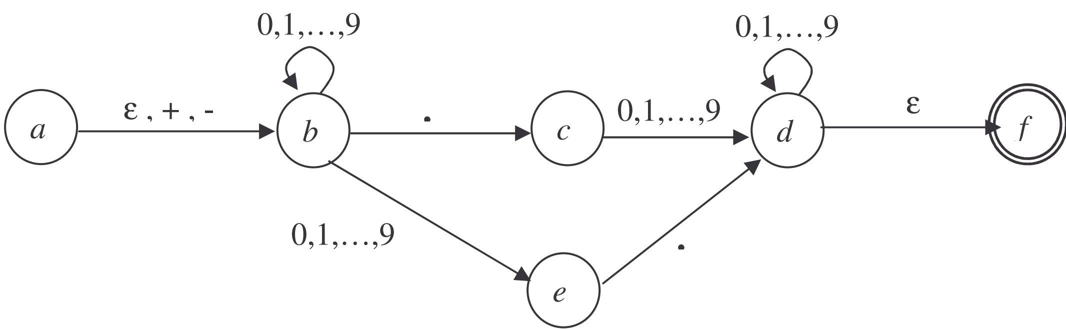


AEF AVEC ε-TRANSITIONS

Ce sont des AEF dans lesquels on autorise que certains arcs portent le symbole ε (la chaine vide) c'est à dire qu'en effectuant la transition entre les 2 états connectés par un tel arc, l'automate ne consomme pas de symbole dans la chaîne courante. Par exemple voici l'AEF qui reconnaît les nombres décimaux ainsi construits :

- un signe optionnel « + » ou « - »
- une suite de chiffres
- un point décimal
- une autre suite de chiffres

L'une des 2 suites de chiffres peut être vide, mais pas les 2.



L'intérêt des ε-transitions est que l'écriture de l'AEF peut être grandement facilitée. L'inconvénient est que le degré de non-déterminisme (et donc l'inefficacité !) augmente considérablement. Par chance, on connaît une méthode pour éliminer ces ε-transitions et donc tirer profit de tous les avantages.

Définition

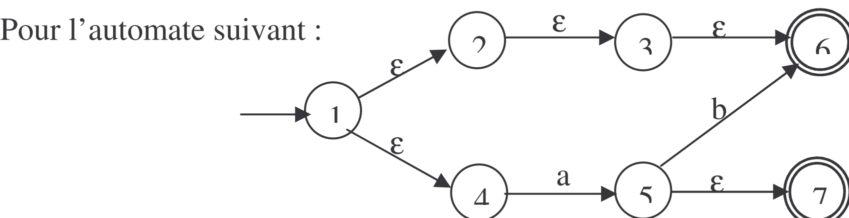
Etant donné un AEF et un état q de cet automate on appelle ***ε-fermeture*** de q l'ensemble définit ainsi :

- ε-fermeture(q) contient q.
- chaque fois que ε-fermeture(q) contient un état p, on lui ajoute tous les états éléments de $\delta(p,\varepsilon)$.

En d'autres termes l'ε-fermeture de q est l'ensemble de tous les états accessibles depuis q par une ε-transition ou une suite d'ε-transitions, y compris q lui-même.

Exemple
Dans l'automate précédent :
- ε-fermeture(a) = $\{a , b\}$
- ε-fermeture(b) = $\{ b\}$; ε-fermeture(c) = $\{c\}$; ε-fermeture(e) = $\{e\}$;
- ε-fermeture(f) = $\{f\}$
- ε-fermeture(d) = $\{d , f\}$

Pour l'automate suivant :



- ε-fermeture(1) = {1,2,3,4,6} ; ε-fermeture(2) = {2,3,6} ; ε-fermeture(3) = {3,6}
- ε-fermeture(4) = {4} ; ε-fermeture(5) = {5,7} ; ε-fermeture(7) = {7}

Elimination des ε-transitions
Etant donné l'AEF A = <V,Q,D,F,δ> comportant des ε-transitions, le procédé ci-dessous permet de construire un automate déterministe équivalent (qui accepte le même langage) sans ε-transitions A' = <V,Q',D',F',δ'> :

- Q' (c'est à dire les états de A') est l'ensemble des sous-ensembles de Q.
  Plus précisément les états accessibles de Q' sont les sous-ensembles ε-fermés de Q :
  $\{S \subset Q \mid S = \varepsilon\text{-fermeture}(S)\}$

Autrement dit, les sous ensembles ε-fermés de Q sont les parties S pour lesquelles toute ε-transition partant d'un état dans S conduit à un état également dans S. Remarquer que ∅ est un état ε-fermé.

- L'état initial de Q' est q'_0 = ε-fermeture(q_0), q_0 étant l'état initial de Q : l'état initial de Q' est obtenu par ε-fermeture de l'état initial de Q (on pet toujours se ramener à un seul état initial dans A en rajoutant des ε-transitions).
- F' est l'ensemble des sous-ensembles fermés de Q qui contiennent au moins un état final de A :
  $F' = \{S \in Q' \mid S \cap F \neq \varnothing\}$
- $\delta'(S,a)$ pour $S \in Q'$ et $a \in V$ est déterminé comme suit :
  - soit $S = \{p_1,p_2,\ldots,p_k\}$
  - calculer $\bigcup_{i=1}^{k} \delta(p_i,a)$. Soit $\{r_1,r_2,\ldots,r_m\}$cet ensemble.

alors : $\delta'(S,a) = \bigcup_{j=1}^{m} \varepsilon\text{-fermeture}(r_j)$

Exemple

Eliminons les ε-transitions de l'AEF A qui reconnaît les nombres décimaux. Soit A' l'AEF déterministe sans ε-transitions :

Les états accessibles de A' sont les sous-ensembles ε-fermés des états de A. On les détermine comme suit :

- On procède comme pour le calcul de l'automate déterministe en remplissant un tableau portant en ligne les états et en colonne les symboles de V, y compris *ε*.
- On ajoute une colonne supplémentaire où l'on range les ε-fermetures des états ainsi déterminés.
- Chaque fois qu'on engendre un nouvel état, on ajoute une ligne en bas du tableau.

		+	–	.	0..9	ε	ε-ferm.
→	*a*	*b*	*b*	-	-	*b*	*ab*
	b	-	-	*c*	*be*	-	*b*
	c	-	-	-	*d*	-	*c*
	d	-	-	-	*d*	*f*	*df*
	e	-	-	*d*	-	-	*e*
*	*f*	-	-	-	-	-	*f*
⇒	*ab*	*b*	*b*	*c*	*be*	*b*	*ab*
	be	-	-	*cd*	*be*	-	*be*
*	*df*	-	-	-	*d*	*f*	*df*
	cd	-	-	-	*d*	*f*	*cdf*
*	*cdf*	-	-	-	*d*	*f*	*cdf*

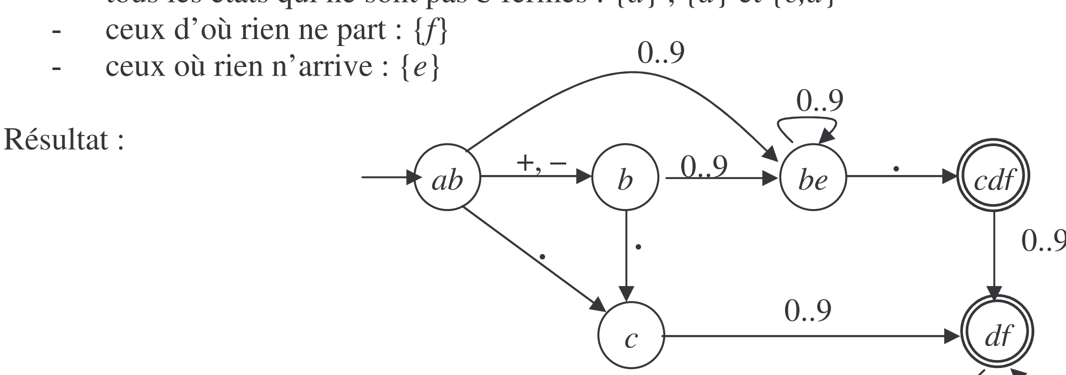
Une fois le tableau complété, dans chacune des colonnes associées aux symboles (sauf *ε*), on remplace les ensembles calculés par leur *ε*-fermetures (trouvées en dernière colonne).

		+	–	.	0..9
→	~~*a*~~	~~*b*~~	~~*b*~~	~~-~~	~~-~~
	b	-	-	*c*	*be*
	c	-	-	-	*df*
	~~*d*~~	~~-~~	~~-~~	~~-~~	~~*df*~~
	~~*e*~~	~~-~~	~~-~~	~~*df*~~	~~-~~
*	~~*f*~~	~~-~~	~~-~~	~~-~~	~~-~~
⇒	*ab*	*b*	*b*	*c*	*be*
	be	-	-	*cdf*	*be*
*	*df*	-	-	-	*df*
	~~*cd*~~	~~-~~	~~-~~	~~-~~	~~*df*~~
*	*cdf*	-	-	-	*df*

L'état initial de A' est l'ε-fermeture de l'état initial de A : {*a*,*b*}noté simplement *ab*.
Les états finaux de A' sont ceux qui contiennent *f*, seul état final de A.

- On supprime :
  - tous les états qui ne sont pas ε-fermés : {*a*} , {*d*} et {*c*,*d*}
  - ceux d'où rien ne part : {*f*}
  - ceux où rien n'arrive : {*e*}

Résultat :



Proposition

Si A est un AEF non déterministe comportant des ε-transitions, et A' l'automate déterministe sans ε-transitions construit comme ci-dessus, alors L(A) = L(A').

Preuve

Soit A = <V,Q,D,F,δ> et A' = <V,Q',D',F',δ'> construit comme ci-dessus. Nous allons monter que $\delta(q_0,w) = \delta'(q'_0,w)$ pour tout mot $w\in$ V, par récurrence sur la longueur de *w*.

- Si |w| = 0, alors w = ε. On sait que $\delta(q_0, \varepsilon)$ = ε-fermeture (q_0) et que q'_0 = ε-fermeture (q_0) par construction. A' étant déterministe, $\delta'(p, \varepsilon) = p$ pour tout état *p* et donc en particulier :
  $\delta'(q'_0, \varepsilon) = q'_0 =$ ε-fermeture (q_0) = $\delta(q_0, \varepsilon)$

- Supposons que la proposition soit vraie pour toute chaîne de longueur *n* et que $|w| = n+1$. On a alors :
  - $w=x.a$ où *a* est le symbole final de *w*
  - et $\delta(q_0, x) = \delta'(q'_0, x) = \{p_1,p_2,\dots,p_k\}$.

Par définition, $\delta(q_0, w)$ est calculé comme suit :

- Soit $\{r_1,r_2,\dots,r_m\}$l'ensemble : $\bigcup_{i=1}^{k} \delta(p_i,a)$.
- $\delta(q_0, w) = \bigcup_{j=1}^{m}$ ε-fermeture(r_j)

Or par construction, $\delta'(\{p_1,p_2,\dots,p_k\},a)$ est défini exactement de la même manière.
Donc $\delta'(q'_0, w)$ qui est la même chose que $\delta'(\{p_1,p_2,\dots,p_k\},a)$ est le même ensemble que $\delta(q_0,w)$. On a donc bien prouvé que $\delta(q_{0,\ w}x) = \delta'(q'_0, w)$ et établi la proposition.

Chapitre III : SYSTEMES DE REECRITURE - GRAMMAIRES

SYSTEMES DE REECRITURE

Définition
Un *système de reécriture* est un couple <V,R> où V est un vocabulaire et R un ensemble fini de couples de chaines construites sur l'alphabet V.
Un élément (a,b) de R est appelé *règle de reécriture* et noté a → b.
La notation a→b|c|…..|d est un raccourci pour a→b ; a→c ; … ; a→d.

Relation de dérivation : définition
Etant donné un système de reécriture <V,R> et 2 chaines x et y de V* on dit que :
- x se *reécrit directement* en y si il existe une règle a → b de R , u et v $\in$ V* tels que x= u a v et y = u b v.

On écrit : $x \Rightarrow y$

On remarque que si $x \Rightarrow y$, plusieurs règles peuvent le permettre. par exemple :
R = {ab→ba ; aab → aba} et aab $\Rightarrow$ aba.

La relation $\Rightarrow$ est une relation sur V*. On définit donc :
- $x \Rightarrow^n y$: x se reécrit en y par application de n règles (x = y si n = 0)
- $x \Rightarrow^* y$: x se reécrit en y par application d'un nombre quelconque (peut être 0) de règles
- $x \Rightarrow^+ y$: x se reécrit en y par application d'au moins une règle

On dira que x se reécrit en y et on appelera dérivation de x toute suite finie $x_0, x_1,\ldots, x_n$, telles que $x = x_0 \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_n = y$. La longueur de la dérivation est n.

Composition de dérivations : proposition
Soit <V,R> un système de reécriture et $u_1,\ldots, u_n$ et $v_1,\ldots, v_n$ des chaines de V*.
Si $u_1 \Rightarrow^* v_1,\ldots, u_n \Rightarrow^* v_n$, alors $u_1u_2\ldots u_n \Rightarrow^* v_1v_2\ldots v_n$
Si $u_1 \Rightarrow^{q1} v_1,\ldots, u_n \Rightarrow^{qn} v_n$, alors $u_1u_2\ldots u_n \Rightarrow^{(q1+ q2+\ldots+ qn)} v_1v_2\ldots v_n$

Preuve
- si $u_1 \Rightarrow^* v_1$ alors $u_1u_2 \Rightarrow^* v_1v_2$: se montre facilement par récurrence sur la longueur de la dérivation
- de même si $u_2 \Rightarrow^* v_2$ alors $v_1u_2 \Rightarrow^* v_1v_2$
- donc si $u_1 \Rightarrow^* v_1$ et $u_2 \Rightarrow^* v_2$ alors $u_1u_2 \Rightarrow^* v_1v_2$

La généralisation au cas de n dérivations est immédiate.

Ainsi la relation $\Rightarrow$ apparaît comme plus générale que la relation →. A partir de maintenant on écrira donc → pour $\Rightarrow$, $\rightarrow^n$ pour $\Rightarrow^n$ et →* pour $\Rightarrow^*$.

Exemples
1. V={a , b , 0, 1 , r , s} R= {ra→s ; rb→s ; sa→s ; sb→s ; s0→s ; s1→s}

   On vérifie que rw→*s ssi w commence par a ou b. Ce système reconnaît les identificateurs composés des lettres {a , b , 0 , 1}

2. V={0 , 1 , + , E} R = {E→0 ; E→1 ; E→E+E}

Pour tout w dans {0,1,+} on a E→*w ssi w est une expression arithmétique construite avec les opérandes 0 et 1 et l'opérateur +.

3. V={0,1,+} R={0+0→0 ; 0+1→1 ; 1+0→1 ; 1+1→0}

   Ce système calcule la somme modulo 2 d'une suite d'entiers binaires : w→*0 ou w→*1

4. V={0,1}   R={10→01}

   Ce système trie les chaines construites sur {0,1} par chiffres croissants

Utilisation
Les systèmes de reécriture vont nous permettre de spécifier des langages, et ceci de 2 façons :

- c'est un procédé de *génération* du langage $L = \{u \in V^* \mid \exists v \in A \text{ tq } v \to^* u\}$ à partir d'une partie A de V*. Ici on engendre toutes les chaines dérivables d'une chaine de A.

  Exemple : V = {a , b} ; ab → aabb ; et A = {ab}.
  On engendre $L = \{u \in V^* \mid ab \to^* u\} = \{a^n b^n \mid n>0\}$

- c'est un procédé de *reconnaissance* du langage $L = \{u \in V^* \mid \exists v \in A \text{ tq } u \to^* v\}$. Ici on reconnaît toutes les chaines qui se dérivent en une chaine de A

  Exemple : V = {a,b,c,d} ; ca → d ; da → c ; d → b ; et A = {b}.
  On reconnaît $L = \{u \in V^* \mid \exists v \in A \text{ tq } u \to^* b\} = \{b\} \cup \{ca^{2n+1}\} \cup \{da^{2n}\}$

Dans la suite, on va s'intéresser à établir des équivalences entre ces types de spécifications des langages : les grammaires (procédé de génération) et les automates (procédé de reconnaissance).

GRAMMAIRES

Définition
On appelle *grammaire* un quadruplet $G = \langle V_T, V_N, S, R\rangle$ où :
- V_T et V_N sont 2 vocabulaires disjoints (respectivement vocabulaire *terminal* et vocabulaire *non terminal*).
- $V = V_T \cup V_N$ est le *vocabulaire* de la grammaire
- S est un élément de V_N appelé *axiome* de la grammaire
- R est un ensemble de règles construites sur V.

Il s'ensuit donc que <V,R> est un système de reécriture

Langage engendré
On appelle *langage engendré* par la grammaire ci-dessus le langage

$$L(G) = \{x \in V_T^* \mid S \to^* x\}$$

On note d'abord que les mots du langage engendré sont des terminaux tous dérivés de S. La nature du langage est précisée selon la forme des règles.

Hiérarchie de Chomsky

En fixant des contraintes particulières sur la structure des parties gauche et droite des règles, on définit des classes spécifiques de grammaires(et des langages engendrés). Voici la classification définie par N. Chomsky :

TYPE 0 : Aucune restriction n'est imposée

TYPE 1 ou contextuelles ou « context sensitive »
Les règles sont de la forme :
$uAv \to u\ x\ v$
avec : $u, v \in (V_T \cup V_N)^*$ $A \in V_N$ et $x \in (V_T \cup V_N)^+$
Attention : le fait que x ne soit pas la chaîne vide est une restriction essentielle.

TYPE 2 ou algébrique ou hors contexte ou « context free »
Les règles sont de la forme :
$A \to u$
avec : $A \in V_N$ et $u \in (V_T \cup V_N)^*$

TYPE 3 ou régulière
Les règles sont de la forme :
$A \to vu$
avec : $A \in V_N$ $v \in V_T$ et $u \in V_T \cup V_N \cup \{\varepsilon\}$

Clairement, plus les restrictions imposées sont fortes, moins il y a de langages qui peuvent être engendrés. Si L0, L1, L2, L3 sont les familles des langages engendrés respectivement par les grammaires de type 0, 1, 2, 3 on a :
$L3 \subset L2 \subset L1 \subset L0$

Exemples
Une grammaire de type 3 (régulière)
$V_T = \{a , b\}$ $V_N = \{S , T\}$ $S \to aS$; $S \to aT$; $T \to bT$; $T \to b$
Le langage engendré est $L = \{a^n b^m \mid m,n>0\} = a^+b^+$

Une grammaire de type 1 (contextuelle)
$V_T = \{a , b , c\}$ $V_N = \{S , B , C\}$
1. $S \to aSBC$
2. $S \to aBC$
3. $CB \to BC$
4. $aB \to ab$
5. $bB \to bb$
6. $bC \to bc$
7. $cC \to cc$

Le langage engendré est $L = \{a^n b^n c^n \mid n>0\}$

Une grammaire de type 2 (hors-contexte)
$V_T = \{a , b\}$ $V_N = \{S\}$
$S \to aSb$ $S \to ab$
Le langage engendré est $L = \{a^n b^n \mid n>0\}$

Une autre grammaire de type 2 (hors-contexte)

$V_T = \{a , b\}$ $V_N = \{S , A , B\}$

1. $S \rightarrow aB$
2. $S \rightarrow bA$
3. $A \rightarrow aS$
4. $A \rightarrow bAA$
5. $A \rightarrow a$
6. $B \rightarrow bS$
7. $B \rightarrow aBB$
8. $B \rightarrow b$

Le langage engendré est $L = \{u \in \{a,b\}^+ \mid u$ contient autant de a que de b$\}$

Chapitre IV : GRAMMAIRES ET LANGAGES REGULIERS

Dans le chapitre précédent nous avons vu comment décrire un langage par des automates d'états finis, c'est à dire par des « machines », puis par des grammaires. Dans ce chapitre nous allons étudier des descriptions algébriques d'une classe particulière de langages (l'algèbre des « expressions régulières » ou ER) et nous allons monter l'équivalence entre ces descriptions : les AEF, les ER et les grammaires de type 3 décrivent la même classe de langages qu'on appelle les langages réguliers. Cependant, les ER et les grammaires de type 3 ont une propriété que les automates n'ont pas: elles sont une représentation déclarative des chaînes qu'on veut accepter. C'est pourquoi elles servent dans les systèmes qui traitent des chaînes de caractères, comme la recherche de textes (ex : le « grep » d'Unix) et les générateurs d'analyseurs lexicaux (ex : LEX).

EXPRESSION REGULIERE

Etant donné un vocabulaire V on appelle *expression régulière* sur V toute expression construite par les opérations d'union (+) , produit (.) et étoile (*) de parties finies de V* et définies ainsi :

- les constantes ε et ∅ sont des expressions régulières qui dénotent respectivement les langages {ε} et ∅.
- Si a est un symbole quelconque, alors a est une expression régulière qui dénote le langage $L(a) = \{a\}$
- Si E et F sont des expressions régulières, alors E+F est une expression régulière qui dénote l'union de L(E) et L(F) : $L(E+F) = L(E) \cup L(F)$
- Si E et F sont des expressions régulières, alors E.F est une expression régulière qui dénote la concaténation de L(E) et L(F) : $L(E.F) = L(E) . L(F) = \{x.y \mid x \in L(E) \text{ et } y \in L(F)\}$
- Si E est une expression régulière, alors E* est une expression régulière qui dénote la fermeture de L(E) : $L(E^*) = [L(E)]^* = \{x_1. x_2\ldots x_n \mid n > 0 \text{ et } x_i \in L(E)\} \cup \{\varepsilon\}$
- Si E est une expression régulière, alors (E) est une expression régulière qui dénote le même langage que E : L((E)) = L(E)

Exemple : pour V = {0,1} :

- l'expression E = (0+1)*.000.(0+1)* représente les chaines contenant 3 zéros consécutifs
- E = ((0+1).(0+1))*+((0+1).(0+1).(0+1))* représente les chaînes de longueur multiples de 2 ou 3.

Propriétés algébriques et simplification

- Précédences : par ordre décroissant l'étoile, la concaténation, l'union
- "+" et "." sont associatifs
- "+" est commutatif
- "+" est distributif sur "."
- "." est distributif sur "+".
- "+" est idempotent : E+E = E
- ∅ est élément neutre pour l'union : ∅+E = E+∅=E
- ε est élément neutre pour la concaténation : ε.E = E.ε = E
- ∅ est absorbant pour la concaténation : ∅.E = E.∅=∅
- (E*)* = E*
- ∅* = ε
- ε* = ε
- (E+F)* = (E*F*)* : donc en particulier (ε+E)* = (ε*E*)* = (εE*)* = (E*)* = E*
- etc.

Définition

Un langage L est *régulier* s'il existe une expression régulière E qui en décrit toutes les chaînes : L = L(E).

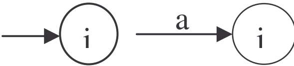
AEF ET LANGAGES REGULIERS

Langage régulier associé à un AEF

Si A est un AEF alors L(A) est un langage régulier

<u>Preuve</u> (par construction)
A chaque état « *i* » de l'automate on associe une variable x_i.

Chaque fois que :

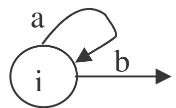


On ajoute l'équation : $x_i = a.\ x_j$. si i n'est pas un état final
et : $x_i = a.\ x_j + \varepsilon$ si i est un état final

En particulier : (i, boucle a) s'écrit $x_i = a.\ x_i$ et (i, état final) s'écrit $x_i = \varepsilon$

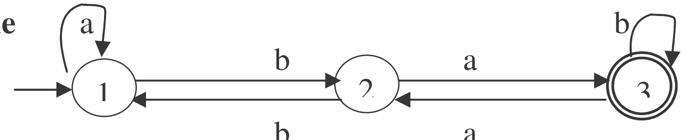
On obtient ainsi un systèmes d'équations sur les langages qui comporte autant de variables que d'états et autant d'équations que de transitions.

On résout ce système en se servant de la *propriété de réduction* suivante:
$x_i = a.\ x_i + b \Leftrightarrow x_i = a^*b$ <u>à condition que *b* ne contienne pas</u> x_i



La solution, c'est à dire l'expression régulière du langage reconnu, est la valeur de x_1 où x_1 est la variable associée à l'état initial.

Exemple



(1) $x_1 = a.\ x_1 + b.\ x_2$
(2) $x_2 = b.\ x_1 + a.\ x_3$
(3) $x_3 = a.\ x_2 + b.\ x_3 + \varepsilon$
----
(4) $x_3 = b^*(a.\ x_2 + \varepsilon)$ propriété de réduction appliquée à (3)
(5) $x_2 = b.\ x_1 + a\ b^*(a.\ x_2 + \varepsilon)$ remplacement dans (2)
(6) $x_2 = ab^*a\ x_2 + ab^* + b\ x_1$ développement de (5)
(7) $x_2 = (ab^*a)^*\ (ab^* + b\ x_1)$ réduction sur (6)
(8) $x_1 = a.\ x_1 + b(ab^*a)^*\ (ab^* + b\ x_1)$ remplacement dans (1)
(9) $x_1 = a.\ x_1 + b(ab^*a)^*\ ab^* + b(ab^*a)^*\ b\ x_1$ développement de (8)
(10) $x_1 = (a + b(ab^*a)^*\ b\ )x_1 + b(ab^*a)^*\ ab^*$ factorisation de (9)
(11) $x_1 = (a + b(ab^*a)^*\ b\ )^*\ b(ab^*a)^*\ ab^*$ réduction de (10)

AEF associé à un langage régulier

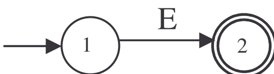
Proposition
Pour tout langage régulier L il existe un AEF A qui reconnaît L : L(A)=L

Preuve
Soit E une expression régulière sur V, L(E) le langage décrit par E et k(E) le nombre d'occurrences de signes +, . et * dans E

Il suffit alors de prouver AEF A qui reconnaît L(E). On montre ceci par récurrence sur k(E). On n'utilisera que des automates « standard » c'est à dire des automates qui ont un seul état initial auquel n'arrive aucune transition (c'est toujours possible).

1. k(E) = 0. Alors $E = \emptyset$ ou $E \in V \cup \{\varepsilon\}$
   - Si $E = \emptyset$, L(E) est reconnu par l'automate

   - Si $E \in V \cup \{\varepsilon\}$, L(E) est reconnu par l'automate



2. Supposons la propriété vraie pour toute expression régulière E tq $k(E) \leq K$ et prouvons la pour F tq $k(F) = K+1$:

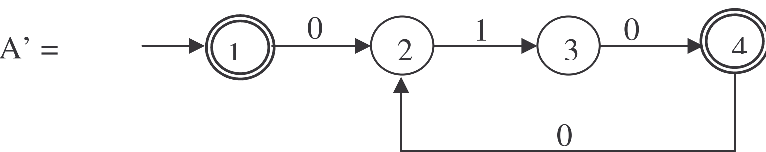
   2.1. G = E*
Soit $A = <V, Q, \{1\}, F, \delta>$ l'AEF qui reconnaît L(E)
   Alors $A' = <V, Q,\{1\}, F\cup\{1\}, \delta\cup\delta'>$ où $\delta'=\{(f, x, q) \mid f \in F \text{ et } (1,x,q)\in \delta\}$ reconnaît L(E*).

   Exemple

   A=

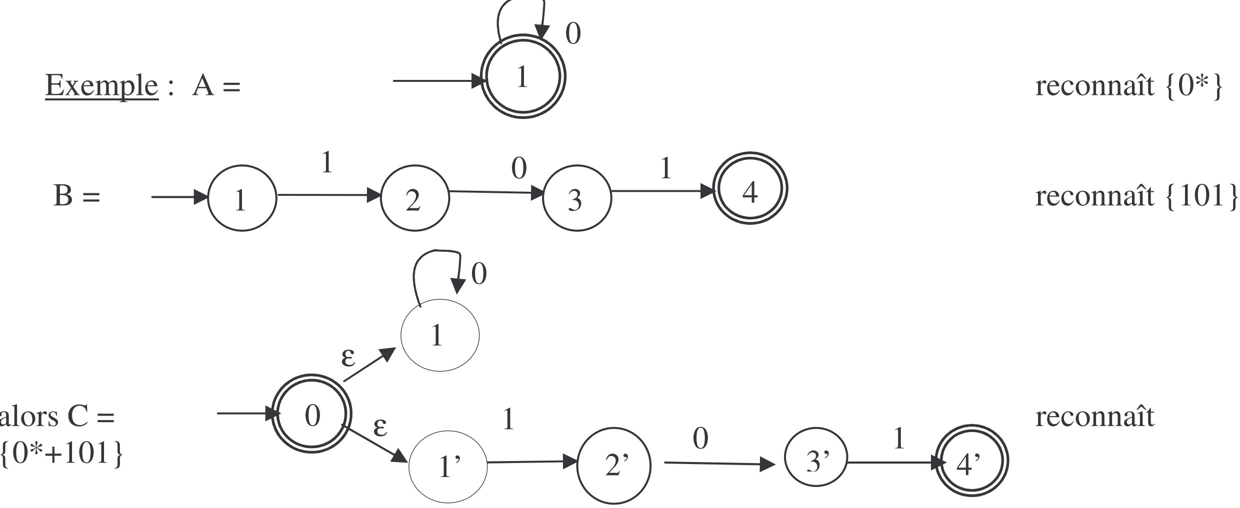
   reconnaît {010}

   A' =



   reconnaît (010)*

   2.2. E = E1+E2
   Si A reconnaît L(E1) et B reconnaît L(E2) on crée un nouvel automate en créant un nouvel état initial avec une ε-transition vers les états initiaux de A et B après avoir renommé les états de façon à ce que A et B n'en aient pas en commun. Cet automate reconnaît L(E1+E2)



2.3. E=E1. E2

$A = < V_1, Q_1, D_1, F_1, \delta_1 >$ reconnaît L(E1)

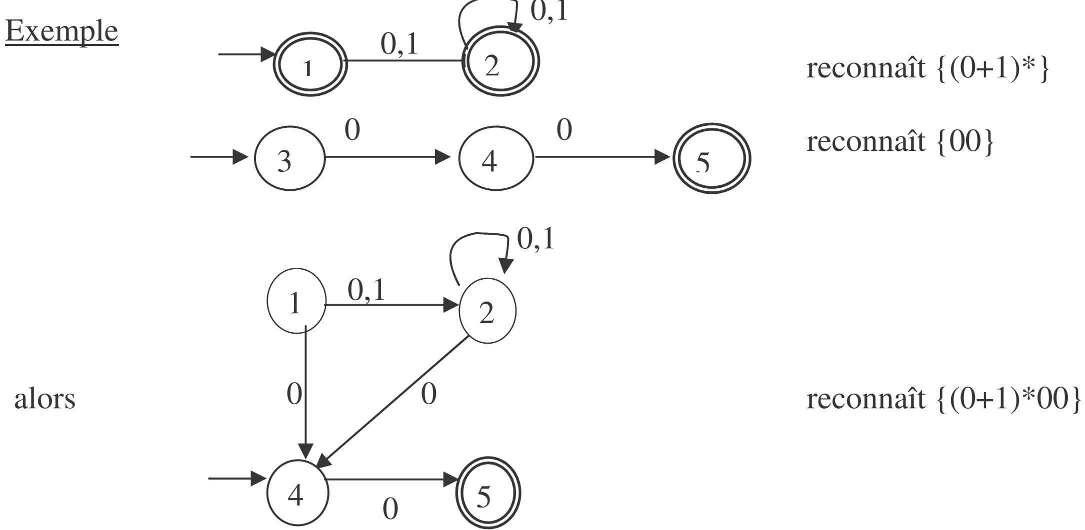
$B = < V_2, Q_2, \{d_2\}, F_2, \delta_2 >$ reconnaît L(E2)

$Q_1 \cap Q_2 = \varnothing$ et $d_2 \neq d_2$

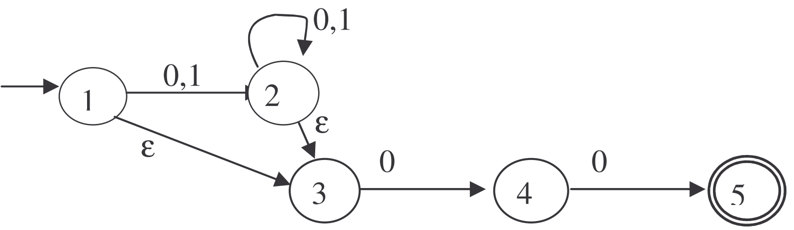
$C = < V, Q, D_1, F, \delta >$ avec

- $V = V_1 \cup V_2$;
- $Q = Q_1 \cup Q_2 - \{d_2\}$;
- $F = F_2$ si $d_2 \notin F_2$ et $F = F_1 \cup F_2 - \{d_2\}$ sinon
- $\delta = \delta_1 \cup \{(q, x, p) \in \delta_2 \mid q \neq d_2\} \cup [\{(q, x, p) \mid q \in F_1 \text{ et } (d_2, x, p) \in \delta_2\}]$

On court-circuite l'état initial de B à partir des états finaux de A.



Plus simplement avec des ε-transitions depuis les états finaux du 1er automate vers l'état initial du 2ème :



GRAMMAIRES REGULIERES ET AEF

AEF associé à une grammaire régulière

Proposition
Pour toute grammaire régulière G il existe un AEF A tel que L(A) = L(G)

Preuve
Etant donné G =<V_T, V_N, S, R> une grammaire régulière
On définit l'AEF A = < V_T, Q, D, F, δ> où:
- le vocabulaire de A est le vocabulaire terminal de G
- l'ensemble des états de A est $Q = V_N \cup \{K\}$ où K est un nouveau symbole $\notin V_N$
- l'ensemble des états finaux de A est F = {S,K} si G contient la règle $S \rightarrow \varepsilon$. Sinon F = {K}
- la relation de transition δ est définie par :

$\delta = \{ (B,a,C) \mid B,C \in V_N,\ a \in V_T$ et $B \rightarrow aC$ est une règle de G $\} \cup$
$\{ (B,a,K) \mid B \in V_N, a \in V_T$ et $B \rightarrow a$ est une règle de G $\}$

Montrons que lorsqu'il accepte une chaine $x= a_1a_2 \ldots a_n \in V^*$, l'automate A simule une dérivation de x par la grammaire G.

Soit $x= a_1a_2 \ldots a_n \in L(G)$ avec n>0. Alors on a :
$S \rightarrow a_1A_1 \rightarrow \ldots \rightarrow a_1a_2 \ldots a_{n-1} A_{n-1} \rightarrow a_1a_2 \ldots a_n$ les A_i étant des non terminaux par application successive des règles de G :
- $S \rightarrow a_1A_1$
- $A_1 \rightarrow a_2A_2$
- ....
- $A_{n-1} \rightarrow a_n$

Par définition de δ on voit que $(S, a_1, A_1) \in \delta$, $(A_1, a_2, A_2) \in \delta$, …,$(A_{n-1}, a_n, K) \in \delta$
Les A_i, K et S étant des états de l'automate, $(S, a_1, A_1)(A_1, a_2, A_2)\ldots(A_{n-1}, a_n, K)$ est un chemin pour x dans A et donc A reconnaît x.

Réciproquement, si A reconnaît x $a_1a_2 \ldots a_n \in V^*$ avec n>0, alors il existe un chemin $(S, a_1, A_1)(A_1, a_2, A_2)\ldots(A_{n-1}, a_n, K)$.
Donc $(S, a_1, A_1) \in \delta$, $(A_1, a_2, A_2) \in \delta$, …,$(A_{n-1}, a_n, K) \in \delta$
Donc $S \rightarrow a_1A_1$; $A_1 \rightarrow a_2A_2$; $A_{n-1} \rightarrow a_n$ sont des règles de G.
Donc $S \rightarrow a_1A_1 \rightarrow \ldots \rightarrow a_1a_2 \ldots a_{n-1} A_{n-1} \rightarrow a_1a_2 \ldots a_n$ est une dérivation de x dans G.
Donc $x \in L(G)$
CQFD

Grammaire régulière associée à un AEF

Proposition
Pour tout AEF A il existe une grammaire régulière G telle L(G) = L(A)

Preuve
Soit A= < V, Q, D, F, δ >. On peut supposer sans perte de généralité que A est déterministe. On définit alors la grammaire G = <V, Q, S, R> où :
- le vocabulaire terminal de G est le même que celui de A
- chaque état de A définit un symbole non-terminal de G (qui n'en a pas d'autre)

- l'axiome S est associé à l'état initial de A
- à chaque transition (q1, x, q2) de A correspond la règle Q1 → x Q2 de G (qui n'en a pas d'autre) où Q1 (resp. Q2) est le non terminal de G associé à l'état q1 (resp. q2) de A.

On poursuit comme pour le théorème précédent.

Conclusion
Des résultats de ce chapitre on peut conclure que :

1. Il y a équivalence entre les langages réguliers (définis par des expressions régulières), les automates d'état finis et les grammaires de type 3.
2. Qu'il est toujours possible de définir un analyseur déterministe pour un langage régulier.
3. Que la famille des langages réguliers est *fermée* pour les opérations binaires d'*union*, de *concaténation* et pour les opérations unaires *étoile* et *complémentaire*.
4. L'ensemble des chaines accepté par un AEF à n états est :
   - non vide ssi l'AEF accepte une chaine de longueur inférieure à n
   - infini ssi l'AEF accepte une chaine de longueur k, $n \leq k \leq 2n$
5. Il y a un algorithme pour dire si deux AEF sont équivalents.
6. Ces deux dernières propriétés sont des propriétés de *décidabilité*. Attention : la propriété 5. n'est pas vraie pour les grammaires de type 0, 1 et 2.

Preuve de la propriété 4

(1) La partie « si » est triviale. Pour l'autre, supposons que l'AEF $A = \langle V,Q,D,F,\delta \rangle$ ayant n états accepte au moins une chaine et soit w une chaine dont la longueur est celle de la plus courte chaine acceptée. Si $|w| < n$ alors la propriété est prouvée. Sinon $|w| \geq n$. Si A accepte w, A ayant n états repasse nécessairement 2 fois par un même état q et on peut écrire $w= w_1w_2w_3$ avec $w_2 \neq \varepsilon$ et $q0 \in D$, $\delta(q0, w_1) = q$, $\delta(q, w_2) = q$ et $\delta(q, w_3) \in F$. Donc A accepte $w_1 w_3$ avec $|w_1 w_3| < |w|$ ce qui contredit l'hypothèse que la plus courte chaine acceptée est de longueur $|w|$.

(2) Si $w \in L(A)$ et $n \leq |w| < 2n$ alors on peut écrire $w = w_1w_2w_3$ avec $w_2 \neq \varepsilon$ et, pour tout i, $w_1w_2^iw_3 \in L(A)$ et donc L(A) est infini.
Par ailleurs, si A accepte une infinité de chaines dont aucune n'est de longueur entre n et 2n-1, soit w une chaine de longueur aussi courte que la plus courte chaine acceptée de longueur $\geq 2n$. Alors on peut écrire $w = w_1w_2w_3$ avec $1 \leq |w_2| \leq n$ et $w_1w_3 \in L(A)$; d'où contradiction.

Remarque : les conditions « l'AEF accepte une chaine de longueur inférieure à n » et « l'AEF accepte une chaine de longueur k, $n \leq k \leq 2n$ » sont toujours vérifiables : il suffit d'énumérer toutes les chaines des longueurs concernées, qui sont en nombre fini.

Preuve de la propriété 5
Soient A1 et A2 des AEF, L1=L(A1) et L2=L(A2).
Alors : $L3 = [L1 \cap L2^C] \cup [L1^C \cap L2]$ est accepté par un certain AEF C.
Or $L3 = [L1 \cup L1^C] \cap [L1 \cup L2] \cap [L2^C \cup L1^C] \cap [L2 \cup L2^C] = [L1 \cup L2] \cap [L1 \cap L2]^C$
Et donc $L3 \neq \emptyset$ ssi $L1 \neq L2$
Alors la propriété précédente permet de dire si oui ou non L1=L2.

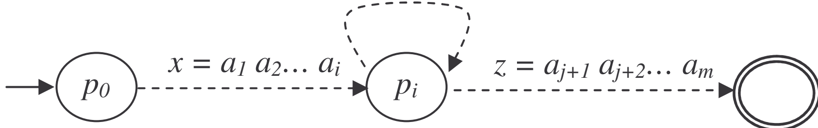
LEMME CARACTERISTIQUE
Ce théorème (connu sous le nom de « pumping lemma ») caractérise les langages réguliers. On l'utilise en pratique pour montrer avec un contre-exemple qu'un langage n'est pas régulier.

Enoncé
Soit L un langage régulier. Alors il existe un entier K dépendant de L tel que toute chaine w de L de longueur $|w| \geq K$ puisse être décomposée en 3 chaines $w = xyz$ telles ques :
(1) $y \neq \varepsilon$
(2) $|xy| \leq K$
(3) Pour tout $n \geq 0$, la chaîne xy^nz est dans L

En résumé, on peut toujours trouver une chaîne non vide y qui ne commence pas trop loin du début de w qui peut être retirée autant de fois qu'on veut, le reste étant toujours dans L.

Preuve
Soit L régulier, A l'AEF déterministe associé et K le nombre d'états de A. Soit une chaîne w de longueur au moins égale à K : $w = a_1.\ a_2\ldots\ a_m$. avec $m \geq K$.
Pour $i = 0, \ldots, m$ soit $p_i = \delta(q_0, a_1a_2...a_i)$ où δ est la fonction de transition de A et $q_0 = p_0$ l'état initial. Puisque $m \geq K$, les p_i ne sont pas tous distincts et il y a donc 2 entiers différents i et $j \leq K$ tels que $p_i = p_j$. Alors on peut écrire :
1. $x = a_1.\ a_2\ldots\ a_i$: x arrive sur p_i une fois (x est vide si $i = 0$)
2. $y = a_{i+1}.\ a_{i+2}\ldots\ a_j$: y part de p_i et y revient (car $p_j = p_i$).
3. $z = a_{j+1}.\ a_{j+2}\ldots\ a_m$: et z est le reste de w. z est vide si $j = m = K$

Voilà ce qui se passe : $y = a_{i+1}\ a_{i+2}\ldots\ a_j$



Ce qui montre que :
(1) y n'est jamais vide ($i \neq j$)
(2) $|xy| = j \leq K$
(3) si l'automate reçoit xy^nz :
   - si $n = 0$ alors A accepte xz et xz est dans L
   - si $n > 0$ alors A consomme x, fait n fois la boucle puis consomme z. Donc A accepte xy^nz qui est bien dans L.

Application
On utilise ce lemme pour montrer qu'un langage L n'est pas régulier. On peut voir les choses sous la forme suivante :
1. On suppose que L est régulier et on prend le nombre K donné par le lemme
2. On choisit une chaîne w de L de longueur au moins égale à K
3. On essaie de décomposer w en xyz avec les contraintes $|xy| \leq K$ et $y \neq \varepsilon$
4. Si on trouve un entier n tel que xy^nz n'est pas dans L, alors le lemme est contredit et L n'est pas régulier

Exemple
L ={ chaînes sur {0,1}* ayant même nombre de 1 et de 0} n'est pas régulier.
1. On suppose que L est régulier. Soit K le nombre donné par le lemme

2. Soit w = 0^K1^K qui est un cas très particulier d'élément de L

3. On décompose w en xyz avec les contraintes $|xy| \leq K$ et $y \neq \varepsilon$. Alors puisque $|xy| \leq K$ et que xy est le début de w :
   - x et y ne contiennent chacun que des 0.
   - x en contient un nombre $< K$ puisque y n'est pas vide.
   - z contient tous les 1.

Donc xz contient moins de 0 que de 1. Ceci contredit le (3) du lemme qui dit que xz est dans L.

Exemple
L ={ $w \in \{1\}^*$ tq $|w|$ est un nombre premier} n'est pas régulier.

1. On suppose que L est régulier. Soit K le nombre donné par le lemme.

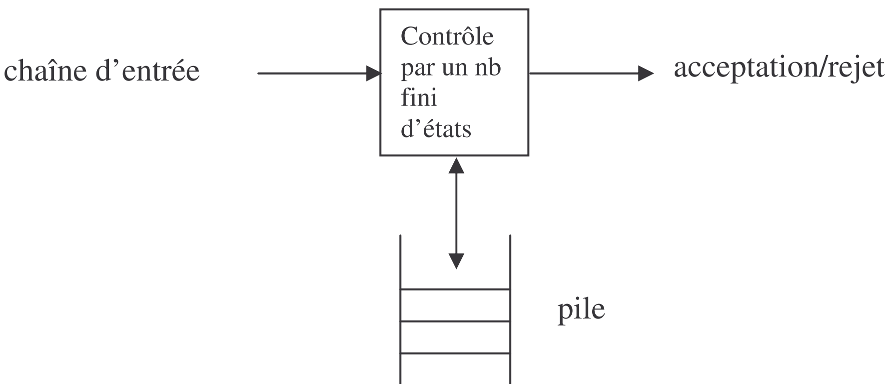
2. Soit p un nombre premier $\geq K+2$ et $w = 1^p$.

3. On décompose w en xyz avec les contraintes $|xy| \leq K$ et $y \neq \varepsilon$. Soit $|y| = m$ et $|xz| = p\text{-}m$. Considérons $xy^{p-m}z$: $|xy^{p-m}z| = |xz| + (p\text{-}m)\,|y| = p\text{-}m + (p\text{-}m)m = (p\text{-}m)(m+1)$ n'est pas un nombre premier car aucun de ces facteurs n'est égal à 1 :
   - $m+1 > 1$ car $m = |y| > 0$ (y n'est pas vide)
   - $p \geq K+2$ et $m = |y| \leq |xy| \leq K$ implique $p\text{-}m > 1$
     (car : $p \geq K+2$, $-m \geq -K$ donc $p - m \geq 2 > 1$)

Ce qui contredit le lemme (3). Donc L n'est pas régulier.

Chapitre V : AUTOMATES A PILE

Un automate à pile (AAP) est essentiellement un AEF non déterministe avec ε-transitions. Il a une propriété supplémentaire, c'est d'être muni d'une pile qui permet de mémoriser un nombre quelconque d'informations et de les retrouver le moment venu. En première approximation, un AAP peut être décrit comme suit :



Un dispositif de « contrôle » lit un à un les symboles de la chaîne d'entrée. L'AAP peut observer le symbole placé au sommet de la pile et de définir ses transitions en fonction de son état courant, du symbole courant de la chaîne lue et du symbole en sommet de pile. Il peut aussi faire une transition « spontanée » en utilisant ε au lieu du symbole lu. Au cours d'une transition l'AAP effectue les opérations suivantes :

1. Consommer le symbole courant de la chaîne lue ou ne pas le consommer s'il s'agit d'une ε-transition.
2. Changer d'état, ou rester dans le même état.
3. Remplacer le symbole de sommet de pile par une chaîne quelconque. Ce peut être par le symbole du sommet de pile, ce qui revient à laisser la pile inchangée. Ce peut être ε ce qui revient à dépiler. Ce peut être un ou plusieurs nouveaux symboles qui sont alors empilé en lieu et place de l'ancien sommet de pile.

DEFINITIONS

Un *automate à pile* est un 7-uplet $A = < K, \Sigma, \Gamma, \delta, q0, Z, F >$

- K : ensemble fini d'états
- $q0 \in K$: état initial
- $F \subset K$: états finaux
- Σ : alphabet fini
- Γ : alphabet de pile
- $Z \in \Gamma$: symbole de fond de pile
- $\delta : K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow$ {parties finies de $K \times \Gamma^*$} : fonction de transition

Une *configuration* de l'automate est un triplet (q,w,a) où

- $q \in K$ est l'état courant
- $w \in \Sigma^*$ est la chaine courante
- $a \in \Gamma^*$ est la pile courante

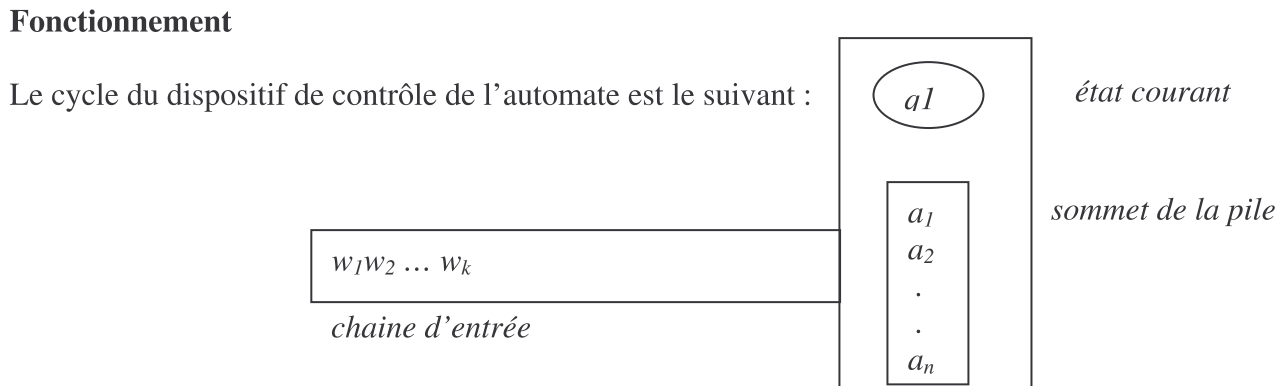
Une *transition* est une relation définie entre 2 configurations par :

$$(q, aw, Xu) \rightarrow (r, w, tu) \text{ ssi } (r, t) \in \delta(q, a, X)$$

On l'interprète comme suit : A la lecture de la lettre *a*, le sommet de pile *X* est remplacé par le mot *t*. Si $t = \varepsilon$ l'effet est de dépiler *X*.

Fonctionnement

Le cycle du dispositif de contrôle de l'automate est le suivant :



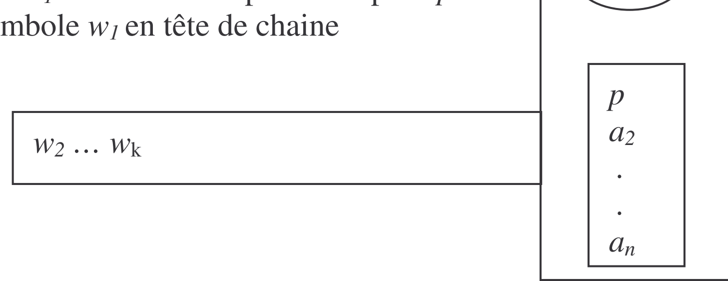
A un instant donné le fonctionnement de l'automate est défini par :
- le symbole en tête de la chaine d'entrée, ici w_1
- l'état courant, ici q_1
- le sommet de la pile courante, ici a_1

ce qui n'est rien d'autre que la configuration $(q_1, w_1 \ldots w_k, a_1 \ldots a_n)$

S'il existe un état q_2 et un symbole de pile p tels que $(q_2, p) \in \delta\, (q_1, w_1, a_1)$
alors la transition correspondante :

$$(q_1, w_1 \ldots w_k, a_1 \ldots a_n) \rightarrow (q_2, w_2 \ldots w_k, p\, a_2 \ldots a_n)$$

s'effectue et l'automate va :
- changer d'état et passer en q_2
- effacer le symbole a_1 en sommet de pile et empiler p.
- consommer le symbole w_1 en tête de chaine



Automate déterministe/non-déterministe

A un triplet de K x $(\Sigma \cup \{\varepsilon\})$ x Γ la fonction de transition δ associe une ensemble de doublets de K x Γ^* . Dans le cas où cet ensemble de doublets est un singleton, l'automate est déterministe. Dans le cas contraire il est non-déterministe.

RECONNAISSANCE/ACCEPTATION

Si, en répétant le processus ci-dessus en partant de l'état initial *q0* et d'une pile ne contenant que le symbole Z , on parvient simultanément à épuiser la chaine d'entrée et la pile, on dit que le mot w est reconnu en mode pile vide, ce qui s'énonce :

Définition

Un mot w est *reconnu* ou *accepté* par pile vide par l'automate A ssi on a : $(q0 , w , Z) \rightarrow^* (q, \varepsilon , \varepsilon)$, où q est un état quelconque.

On appelle **V** (**A**) le *langage engendré par pile vide* par A.
V (**A**) = {w $\in \Sigma^*$ | $\exists$ q $\in$ K tq (q0 , w , Z) →* (q, ε , ε)}

Si, en répétant le processus ci-dessus en partant de l'état initial q0 et d'une pile ne contenant que le symbole Z , on parvient à épuiser la chaine d'entrée en se retrouvant dans un état final q, on dit que le mot w est reconnu en mode état final, ce qui s'énonce :

Un mot w est *reconnu* ou *accepté* par état final par l'automate A ssi on a : (q0 , w , Z) →* (q, ε , a) , où q est un état final.
On appelle **F** (**A**) le langage engendré par état final par A.
F (**A**) = {w $\in \Sigma^*$ | $\exists$ q $\in$ F et a $\in \Gamma^*$ tq (q0 , w , Z) →* (q, ε , a)}

EQUIVALENCE MODE PILE VIDE-MODE ETAT FINAL

Théorème
Un langage L est reconnaissable par un AAP en mode état final ssi L est reconnaissable par un AAP en mode pile vide.

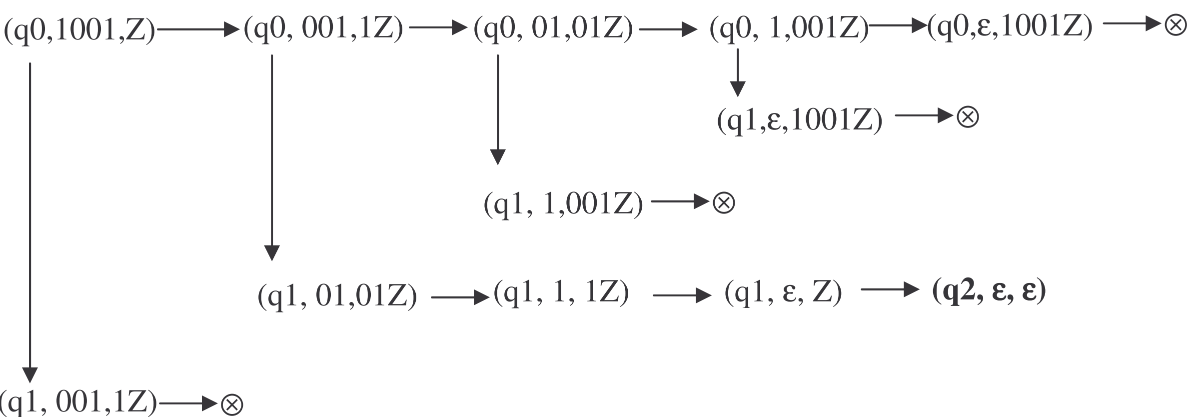
Esquisse de la démonstration
Soit L un langage reconnaissable par un AAP A en mode état final. Pour trouver A' reconnaissant L en mode pile vide on ajoute un nouveau symbole de pile S (empilé avant de démarrer le processus de reconnaissance et qui restera dans la pile jusqu'à la fin) et un nouvel état q_S qui sera atteint après un état final de A et qui servira à vider la pile en posant $\delta(q_S , \varepsilon , S) = \{(q_S , \varepsilon)\}$
Réciproquement, si L est un langage reconnaissable par un AAP A en mode pile vide. Pour trouver A' reconnaissant L en mode état final on ajoute un état final qui ne sera atteint que lorsque la pile de A devient vide.

Exemple : Automate reconnaissant L ={*ww~* | *w* ∈ (0+1)*} en mode pile vide
Soit le langage L = {*ww~* | *w* ∈ (0+1)*} qui est le langage des palindromes pairs sur {0,1}. Informellement, un AAP reconnaissant L peut être construit comme suit :

1. On part d'un état *q0* qui représente la situation où on n'a pas encore trouvé le « milieu », c'est à dire qu'on n'a pas encore atteint la fin de la chaîne *w*. tant qu'on est dans cet état, on lit les symboles un à un et on les range dans la pile.
2. Mais à tout moment, on peut penser qu'on a atteint le milieu, donc qu'on a épuisé *w*. A ce moment *w* est sur la pile, à l'envers (le premier symbole de *w* est au fond de la pile, le dernier au sommet). On indique ce choix en passant spontanément à l'état *q1*. Comme l'automate est non déterministe, on garde les 2 options :
   - l'une selon laquelle on n'a pas encore épuisé *w* : on reste en *q0*, on continue à lire et à emplier les symboles,
   - l'autre selon laquelle on n'a pas encore épuisé *w* : on est en *q1*
3. Une fois en *q1*, on compare le symbole lu avec le sommet de pile. Si ce sont les mêmes, on dépile, on lit un nouveau symbole et on continue. Sinon, c'est qu'on a pris la mauvaise option, *w* n'était pas épuisée. Cette branche ne mène nulle part et la « magie » du non déterminisme fait qu'on se retrouve dans une branche correspondant à l'autre option.
4. Si on finit par vider la pile, on a trouvé *w* suivie de *w~*. On accepte la chaîne lue jusque là.

A = < K, Σ , Γ , δ , q0 , Z , F >

- K= {q0,q1,q2}
- Σ = {0,1}
- Γ = {Z}
- F = {q2}
- δ(q0,0, Z) = {(q0,0Z) , (q1,0Z)}
- δ(q0,0,0) = {(q0,00) , (q1,00)}
- δ(q0,0,1) = {(q0,01) , (q1,01)} *option 1 : on n'a pas épuisé w : on lit et on empile*
- δ(q0,1,Z) = {(q0,1Z) , (q1,1Z)} *option 2 : considérant qu'on a épuisé w, on passe en q1*
- δ(q0,1,0) = {(q0,10) , (q1,10)}
- δ(q0,1,1) = {(q0,11), (q1,11)}
- δ(q1, 0,0) = {(q1, c)} *option 2 : tant que le symbole lu et le sommet de pile*
- δ(q1, 1,1) = {(q1, ε)} *sont les mêmes, on continue*
- δ(q1, ε,Z) = {(q2, ε)} *Et là tout va bien*



Exemple : Automate reconnaissant L ={$a^n b^n$ | n ≥ 0} en mode pile vide

A = < K, Σ , Γ , δ , q0 , Z , F >

- K= {q0,q1,q2}
- Σ = {a , b}
- Γ = {Z}
- δ(q0,a,Z) = {(q1,Z)} *empile le 1er « a »*
- δ(q1,a,Z) = {(q1,ZZ)} *et les suivants et les compte avec des Z*
- δ(q1,b,Z) = {(q2,ε)} *dépile le 1er « b »*
- δ(q2,b,Z) = {(q2,ε)} *et les suivants en comptant les Z*

symbole courant	↓	↓	↓	↓	↓	↓	↓
chaine courante	aaabbb	aaabbb	aaabbb	aaabbb	aaabbb	aaabbb	aaabbb
pile	Z	Z	ZZ	ZZZ	ZZ	Z	ε
état	q0	q1	q1	q1	q2	q2	q2

Chapitre VI : GRAMMAIRES ET LANGAGES HORS-CONTEXTE

Les langages hors-contexte sont une classe de langages plus large que celle des langages réguliers. Ces langages peuvent être décrits par une notation récursive très puissante : les grammaires hors-contexte (GHC) ou de type 2. Historiquement, leur importance vient de ce qu'elles permettent de définir la syntaxe des principaux langages de programmation. Dans des utilisations plus récetes, elles servent à décrire des formats de documents (DTD) pour la « norme » XML d'échange d'information sur le web. Ce chapitre reprend les principales propriétés de ces grammaires et conclut sur l'existence de formes normales auxquelles on peut directement rattacher les automates à pile. Nous montrerons que AAP et langages hors contexte sont équivalents comme nous l'avons vu pour les AEF et les langages réguliers.

EXEMPLES INFORMELS

Exemple 1
Considérons le langage des palindromes (mots qui se lisent aussi bien de gauche à droite que de droite à gauche), par exemple « esoperesteicietserepose » (Esope reste ici et se repose). Pour simplifier, on considérera les palindromes construits sur {0 , 1}.

On vérifie aisément que L = $\{w.a.w\sim \mid w \in \{0,1\}^*, a \in \{0,1, \varepsilon\}\}$ n'est pas régulier grâce au lemme fondamental : soit $w = 0^{K+1}10^{K+1}$ où K est l'entier défini par le lemme.
Si L est régulier, alors on peut écrire $w = xyz$ où $|xy| \leq K$ implique que y contient un ou plusieurs '0' parmi les $K+1$ du début de w. Alors xz qui appartient aussi à L si L est régulier a moins de '0' à gauche qu'à droite du '1' et ne peut donc pas être un palindrome.

Par ailleurs, on peut définir récursivement les palindromes comme suit :
- ε, 0, et 1 sont des palindromes
- si w est un palindrome, 0w0 et 1w1 sont aussi des palindromes

Une grammaire hors contexte est précisément un formalisme qui exprime ce type de définition récursive pour des langages. Une telle grammaire comporte des non-terminaux qui représentent des classes de chaines. Dans notre exemple, un seul non terminal est nécessaire, c'est celui qui représente l'ensemble des palindromes et sera également par conséquent l'axiome S. La définition récursive s'exprime par des règles comme suit :
S → ε   S → 0   S → 1   S → 0S0   S → 1S1

Exemple 2
Intéressons nous maintenant au langage des expressions aritmétiques d'un langage de programmation classique. On ne retiendra que 2 opérateurs, '+' et '*' et les identificateurs autorisés seront construits sur {a,b,0,1}.

La grammaire de ce langage nécessite 2 non-terminaux : E pour représenter les expressions et I pour les identificateurs. Le langage I est régulier et correspond à l'expression régulière :

(a+b)(a+b+0+1)*

La grammaire complète s'écrit ainsi (E est l'axiome) :
E → I   E → E+E   E → E*E   E → (E)
I → a   I → b   I → Ia   I → Ib   I → I0   I → I1

RAPPELS ET PREMIERES DEFINITIONS

Une grammaire $G = < V_T, V_N, S, R >$ est hors contexte si toutes ses règles sont de la forme :

$A \rightarrow w$ où $A \in V_N$ et $w \in (V_T \cup V_N)^*$

Le langage engendré par G est $L(G) = \{w \in V_T^* \mid S \rightarrow^* w\}$. C'est un langage hors contexte.

Lemme fondamental

Si $u_1u_2 \rightarrow^k v$ alors $v = v_1v_2$ avec $u_1 \rightarrow^{k1} v_1$; $u_2 \rightarrow^{k2} v_2$ et $k_1+k_2 = k$

Preuve : par récurrence sur k
- si $k = 0$ c'est évident
- si le lemme est vrai pour toutes les dérivations d'ordre inférieur à k il est encore vrai pour k :

si $u_1u_2 \rightarrow^k v$ alors $u_1u_2 \rightarrow^{k-1} w \rightarrow v$
par hypothèse de récurrence $w = w_1w_2$ avec $u_1 \rightarrow^{k1} w_1$; $u_2 \rightarrow^{k2} w_2$ et $k_1+k_2 = k-1$ et $w_1w_2 \rightarrow v$
de même : $v = v_1v_2$, $w_1 \rightarrow^{h1} v_1$, $w_2 \rightarrow^{h2} v_2$ et $h_1+h_2 = 1$
Donc $u_1 \rightarrow^{k1+h1} v_1$, $u_2 \rightarrow^{k2+h2} v_2$ et $(k_1+h_1) + (k_2+h_2) = k$. CQFD

Application
Ce lemme est utilisé pour montrer qu'un langage est hors-contexte. Exemple : le langage de Dick D_n^* sur n paires de parenthèses. Ce langage est défini sur le vocabulaire terminal :
$Z = Z_n \cup \underline{Z}_n$ où :
- $Z_n = \{ z_i \mid 1 \le i \le n\}$ et $\underline{Z}_n = \{ \underline{z}_i \mid 1 \le i \le n\}$
- les éléments du langage sont les chaînes « bien parenthésées », c'est à dire celles qui se réduisent au mot vide par une succession d'effacements de facteurs de la forme $z_i \underline{z}_i$

Par exemple avec n = 3 on peut prendre :
- $z_1 =$ (; $\underline{z}_1 =$) ; $z_2 =$ [; $\underline{z}_2 =$] ; $z_3 =$ { ; $\underline{z}_3 =$ } ;
- (([(( ))]{( )})) et ((( ))) [{ }] appartiennent au langage, mais pas (([(( )){] ( )}))

Montrons que ce langage est engendré par la grammaire dont les règles sont :
$\{ S \rightarrow z_i S \underline{z}_i S \mid 1 \le i \le n \} \cup \{S \rightarrow \varepsilon\}$.

1. $D_n^* \subset L(S)$. Soit donc $u \in D_n^*$ et $|u| = p$. On procède par récurrence sur p, longueur de u.
- si p=0 alors $u = \varepsilon \in L(S)$
- hypothèse de récurrence : $v \in D_n^*$ et $|v| \le p \Rightarrow v \in L(S)$

Soit alors $u \in D_n^*$ et $|u| = p+1$. Alors u commence par un certain z_i et il existe une décomposition de u : $u = z_i u_1 \underline{z}_i u_2$ avec u_1 et $u_2 \in D_n^*$. Donc u_1 et $u_2 \in L(S)$. Il existe donc une dérivation $S \rightarrow^* u_1$ et une dérivation $S \rightarrow^* u_1$ et donc :
$S \rightarrow z_i S \underline{z}_i S \rightarrow^* z_i u_1 \underline{z}_i S \rightarrow^* z_i u_1 \underline{z}_i u_2 = u$ est une dérivation de u et $u \in L(S)$

2. $L(S) \subset D_n^*$. Soit $u \in L(S)$. Donc $S \rightarrow^k u$. On procède par récurrence sur la longueur k de la dérivation de u.
- si k = 1 alors $u = z_i\underline{z}_i \in D_n^*$ ou $u = \varepsilon \in D_n^*$
- hypothèse de récurrence : $S \rightarrow^k v$ et $k \le p \Rightarrow v \in D_n^*$

Soit u tel que $S \rightarrow^{p+1} u$. Alors il existe v tel que $S \rightarrow v \rightarrow^p u$ avec $p \neq 0$. Alors $v = z_i S \underline{z}_i S$ et donc : $z_i S \underline{z}_i S \rightarrow^p u$.
D'après le lemme fondamental, on a : $u = z_i u_1 \underline{z}_i u_2$ avec $S \rightarrow^{p1} u_1$; $S \rightarrow^{p2} u_2$ et $p_1 + p_2 = p$.
Par hypothèse de récurrence, u_1 et $u_2 \in D_n^*$ et donc se réduisent à ε.
Alors : $u = z_i u_1 \underline{z}_i u_2$ se réduit à $z_i \underline{z}_i$ qui se réduit à ε et donc $\in D_n^*$. CQFD

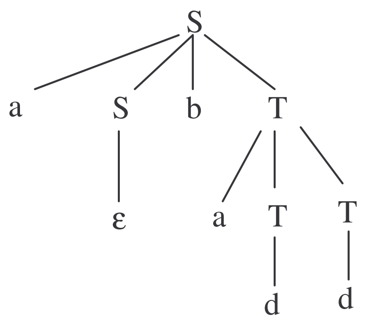
ARBRES DE DERIVATION - AMBIGUITE

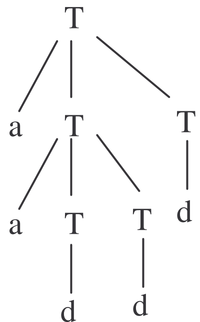
Définition
Si G = < V_T, V_N, S, R > est une grammaire hors contexte on appelle *arbre de dérivation* de G un arbre dont :
- la racine est S
- les nœuds sont dans $V_T \cup V_N \cup \{\varepsilon\}$
- tout sommet a ayant k fils $s_1,\ldots, s_k$ est tel que : $a \in V_N$ et $a \rightarrow s_1\ldots s_k$ est une règle de G
- toute feuille est un élément de V_T

Si $a_1\ldots a_n$ est la suite ordonnée des feuilles de l'arbre obtenue par un parcours en profondeur d'abord et de gauche à droite, alors
$S \rightarrow^* a_1\ldots a_n$
l'arbre en question est un arbre de dérivation du mot $a_1\ldots a_n \in L(G)$

Exemple

S→aSbT S→ cS S→dT S→ε T→aTT T→d





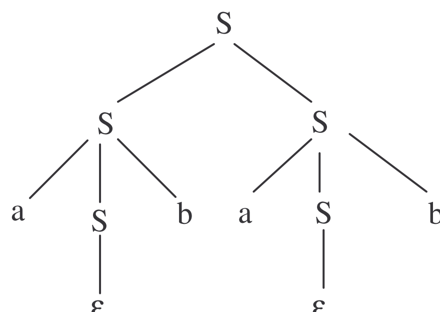
est l'arbre de dérivation de abadd

est l'arbre de dérivation de aaddd

Dérivation gauche (resp droite)
Soit A un non terminal et w tel que A→*w. Une dérivation de A en w est dite *gauche* (resp. *droite*) si w est obtenu à partir de A en appliquant les règles de la grammaire et en dérivant en premier toujours le non-terminal le plus à gauche (resp. droite)

Exemple

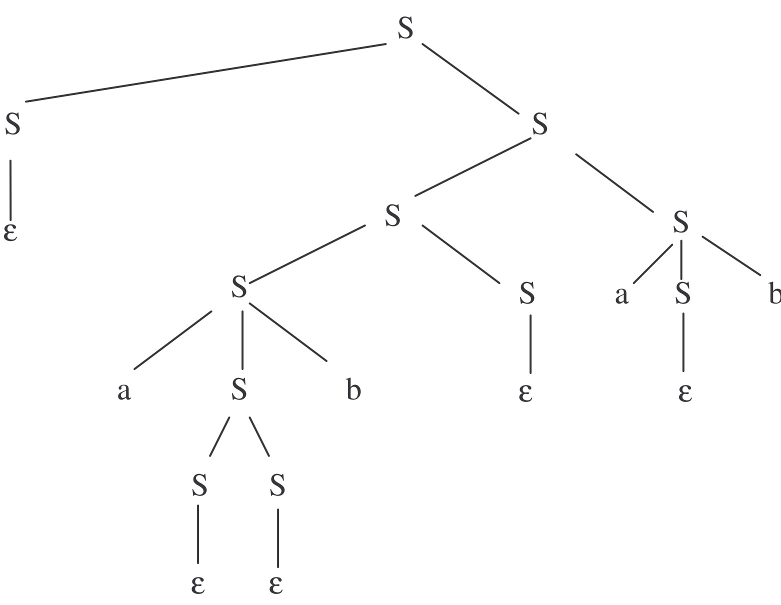
S→SS S→aSb S→ε
La dérivation : S → SS → aSbS → abS → abaSb → abab est gauche



Ambiguité

Une grammaire est dite *ambiguë* s'il existe 2 arbres de dérivation distincts pour un même mot w.

Exemple



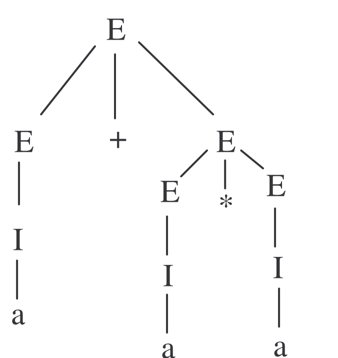
S→SS→S→SS→SSS→aSbSS→aSSbSS→aSbSS→abSS→abS→abaSb→abab
est une autre dérivation de abab

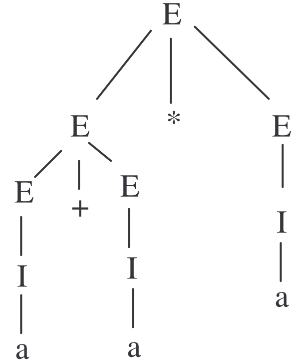
Eliminer les ambiguités

Il n'y a malheureusement pas d'algorithme général qui détecte une grammaire ambiguë, ni pour lever l'ambiguité d'une grammaire. Il y a même des GHC qui sont intrinsèquement ambigues. En pratique, la situation n'est pas si désespérée et pour des constructions comme on les trouve dans les langages de programmation on dispose de quelques techniques pour enlever l'ambiguité. Illustrons cela avec la grammaire des expressions arithmétiques :

E → I	E → E+E	E → E*E	E → (E)		
I → a	I → b	I → Ia	I → Ib	I → I0	I → I1

Cette grammaire est ambiguë comme le montre la dérivation de la chaine a+a*a





E→E+E→I+E→a+E→a+E*E→a+I*E
→ a+a*E → a+a*I→a+a*a

E→E*E→E+E*E→I+E*E→a+E*E
→a+I*E→a+a*E→a+a*I→a+a*a

Cette grammaire est ambiguë pour 2 raisons :

- Les priorités des opérateurs ne sont pas respectées. La 1ère dérivation regroupe le '*' avant le '+'. C'est correct. La 2ème dérivation regroupe le '+' avant le '*'. Ce n'est pas correct. On doit forcer la 1ère dérivation et empêcher la 2ème.
- Des opérateurs identiques qui se suivent peuvent être regroupés aussi bien de droite à gauche que de gauche à droite. Ici encore il faut forcer le regroupement de gauche à droite.

Le moyen de forcer les priorités est d'introduire de nouveaux non-terminaux, chacun représentant les sous-expressions qui ont la même précédence. En particulier :

- un *facteur* est une expression qui ne peut pas être recomposée par un opérateur adjacent, '+' ou '*'. Les seuls facteurs sont :
  - les identificateurs : on ne peut pas séparer les lettres d'un identificateur par un opérateur
  - les expressions entre parenthèses dont le rôle est précisément d'isoler ce qui est à l'intérieur de tout opérateur extérieur.
- un *terme* est une expression qui ne peut pas être recomposée par l'opérateur '+'. Dans notre exemple, un terme est le produit de un ou plusieurs facteurs

De ceci il s'ensuit qu'une expression est une somme de un ou plusieurs termes.

La grammaire obtenue est alors :

E → T	E → E+T				
T → F	T → T*F				
F → I	F → (E)				
I → a	I → b	I → Ia	I → Ib	I → I0	I → I1

Langage intrinsèquement ambigu

Un langage est intrinsèquement ambigu si toutes ses grammaires le sont. En voci un :

L = $\{a^n\ b^n\ c^m\ d^m \mid n > 0 \text{ et } m > 0\} \cup \{a^n\ b^m\ c^m\ d^n \mid n > 0 \text{ et } m > 0\}$, ensemble des chaines $a^+b^+c^+d^+$ où il y a autant de *a* que de *b* et autant de *c* que de *d* ou autant de *a* que de *d* et autant de *c* que de *b*.L est HC et sa grammaire est :

S → *AB* | *C*
A → *aAb* | *ab*
B → *cBd* | *cd*
C → *aCd* | *aDb*
D → *bDc* | *bc*

L'argument pour montrer que L est ambigu est le suivant :

- Il faut un groupe de règles pour engendrer la 1ère moitié du langage (les *a* et les *b* ensemble et les *c* et les *d* ensemble)
- Il faut un autre groupe de règles pour engendrer la 2ème moitié du langage (les *a* et les *d* ensemble et les *b* et les *c* ensemble)
- Une chaîne de L ayant même nombre de *a,* de *b,* de *c* et de *d* peut être engendrée des 2 façons ci-dessus.

FORMES NORMALES DES GRAMMAIRES H.C.

Dans tout ce qui suit on considère une grammaire hors contexte $G = < V_T, V_N, S, R >$

Grammaires équivalentes
Deux g.h.c. $G = < V_T, V_N, S, R >$ et $G' = < V_T, V'_N, S', R' >$ sont dites (*fortement*) *équivalentes* ssi $L_G(S) = L_{G'}(S')$

Grammaire réduite
G est *réduite* si :
Tous ses non terminaux sont productifs : pour tout $A \in V_N$ $L(A) \neq \emptyset$
Tous les non terminaux sont accessibles à partir de S : pour tout $A \in V_N$, $A \in L(S)$

Proposition
Pour toute grammaire h.c. G il en existe une autre G' équivalente à G et réduite.

Méthode de réduction

1. Eliminer les symboles non productifs = $\{A \mid L(A) = \emptyset\}$

1. Construire la suite :
   - $U_0 = V_T$
   - $U_i = U_{i-1} \cup \{A \in V_N \mid \exists\, v \in U^*_{i-1}$ tq $A \rightarrow v$ est une règle de **G**$\}$ qui converge vers U
2. Eliminer les règles qui contiennent un élément qui n'est pas dans U (sauf l'axiome)

2. Eliminer les symboles inaccessibles c'est à dire ceux qu'on ne peut atteindre à partir de S

1. Construire la suite :
   - $W_0 = \{S\}$
   - $W_i = W_{i-1} \cup \{ A \in V_N \mid \exists\, B \in W^*_{i-1}$ et $u , v \in V^*$ tq $B \rightarrow uAv$ est une règle de **G** $\}$ qui converge vers W

2. Eliminer les règles dont le membre gauche n'est pas dans W

Exemple
Soit la grammaire G construite sur le vocabulaire terminal {a,b} et le vocabulaire non terminal {S,B,C,D} et les règles suivantes :

$S \rightarrow aB$	$S \rightarrow bBC$	$S \rightarrow abBS$	$S \rightarrow CD$
$B \rightarrow aC$	$B \rightarrow bB$	$B \rightarrow a$	
$C \rightarrow aC$	$C \rightarrow bC$		
$D \rightarrow aB$	$D \rightarrow bS$	$D \rightarrow a$	

1. Elimination des symboles non productifs

- $U_0 = \{a,b\}$ – $U_1 = \{a,b,B,D\}$ – $U_2 = \{a,b,B,S,D\}$ – stabilité
- Les règles restantes sont :

$S \rightarrow aB$	$S \rightarrow abBS$	
$B \rightarrow bB$	$B \rightarrow a$	
$D \rightarrow aB$	$D \rightarrow bS$	$D \rightarrow a$

2. Elimination des symboles inaccessibles

- $W_0 = \{S\}$ – $W_1 = \{S,B\}$– stabilité
- Les règles restantes sont :

  S→aB S→abBS
  B→bB B→a

Grammaire propre

Une grammaire hors contexte G est dite *propre* si elle ne contient aucune règle de la forme :

- A → ε (ε-règles)
- A → B (I-règles)

où A et B $\in V_N$

Proposition

Pour toute grammaire h.c. G il en existe une autre G' réduite et propre équivalente à G au mot vide près

Preuve par construction

Après avoir réduit G comme ci-dessus on élimine successivement les ε-règles puis les I-règles comme montré ci-dessous

A) Elimination des ε-règles

1. Construire la suite :
   - $U_0 = \{ A \in V_N \mid A \rightarrow \varepsilon$ est une règle de **G** $\}$
   - $U_i = U_{i-1} \cup \{ A \in V_N \mid \exists\, v \in U^*_{i-1}$ tq $A \rightarrow v$ est une règle de **G** $\}$

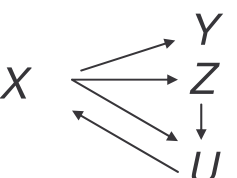
   qui converge vers U
2. Poser :
   - $\sigma(A) = \{ A, \varepsilon \}$ si $A \in V_N$ et $A \in U$
   - $\sigma(A) = \{ A \}$ si $A \in V_N$ et $A \notin U$
   - $\sigma(a) = \{ a \}$ si $a \in V_T$
   - $\sigma(uv) = \{ \sigma(u)\, \sigma(v) \}$ si $|uv| > 1$
3. Remplacer chaque règle $A \rightarrow u$ par l'ensemble des règles $\{A \rightarrow w \mid w \in \sigma(u) \}$
4. Supprimer les règles $A \rightarrow \varepsilon$

B) Elimination des I-règles

1. Regrouper les non-terminaux en classes : A et B sont dans la même classe chaque fois que $A \rightarrow^* B$ et $B \rightarrow^* A$.

   Nommer chaque classe : A^c = classe de A

   Exemple : X → Y | Z | U U → XZ → U
   donne les classes : $X^c = \{X,Z,U\}$et $Y^c = \{Y\}$



2. Poser :
   - $\varphi(A) = A^c$ si $A \in V_N$
   - $\varphi(a) = a$ si $a \in V_T$
   - $\varphi(uv) = \varphi(u)\, \varphi(v)$ si $|uv| > 1$
3. Remplacer chaque règle $A \rightarrow w$ par la règle $\varphi(A) \rightarrow \varphi(w)$ et supprimer les règles $X \rightarrow X$

4. Chercher les éléments maximaux pour la relation définie par :
   A < B ssi A→ B.

   Exemple :

	X → Y ǀ Z ǀ U ǀ V
	Y → U Z → U
	V → U
donne	U comme élément maximal

- Pour chaque élément maximal B, remplacer la règle A → B par l'ensemble de règles {A → w ǀ B → w est une règle de **G**}
- Recommencer 5. jusqu'à la stabilité

Exemple

S→ AB	S→CD	S→E
A→AA	A→D	
B→aB	B→C	
C→B	C→D	C→EaC
D→cD	D→ε	
E→D	E→b	

<u>Elimination des ε-règles</u>

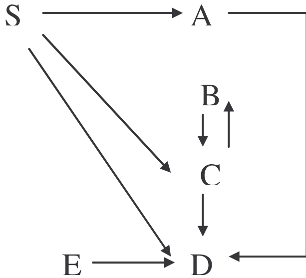
- U_0 = {D} ; U_1 = {D,A,E,C} ; U_2 = {D,A,E,C,B,S}

La grammaire devient :

S→A	S→AB	S→C	S→D	S→CD	~~S→ε~~	
A→AA	A→A	A→D	~~A→ε~~			
B→a	B→aB	B→C	~~B→ε~~			
C→B	~~C→ε~~	C→D	C→EaC	C→aC	C→Ea	C→a
D→cD	D→c	~~D→ε~~				
E→D	~~E→ε~~	E→b				

<u>Elimination des I-règles</u>

Classes :
{S} ; {A} ; {B,C} ; {D} ; {E}



La grammaire devient :

S→A	S→AB	S→B	S→D	S→BD	S→E
A→AA	~~A→A~~	A→D			
B→a	B→aB	~~B→B~~	B→D	B→EaB	B→Ea
D→cD	~~D→D~~	D→c			
E→D	E→b				

Elements maximaux

{D}

La grammaire devient :

S→A	S→AB	S→B	S→cD	S→c	S→BD	S→E
A→AA	A→cD	A→c				
B→a	B→aB	B→cD	B→c	B→EaB	B→Ea	
D→cD	D→c					
E→cD	E→c	E→b				

Et on recommence jusqu'à obtenir la grammaire propre :
S → AB | BD | CD | c | b | AA | aB | a | EaB | Ea
A → AA | cD | c
B → aB | a | cD | c | EaB | Ea
D → cD | c
E→ cD | c | b

Forme normale de Greibach

Une grammaire hors contexte G est sous *forme normale de Greibach* si :

- elle est réduite et propre
- le membre droit de chacune de ses règles commence par un terminal
- à part le 1er symbole, le membre droit de chacune de ses règles ne contient que des non-terminaux

Proposition
Pour toute grammaire h.c. G il en existe une autre G' sous forme normale de Greibach équivalente à G au mot vide près

Méthode de construction
La 1ère condition a été vue précédemment, la 3ème est triviale.
Reste à éliminer les règles X → Xu et les règles X → Yu.

Pour cela, on ordonne arbitrairement les non-terminaux. On les prend dans l'ordre et pour chacun (désigné par A ci-dessous) on effectue à la suite les 2 transformations suivantes

A) Elimination des règles A →Au (ou récursives à gauche)

Soient {A → A u_1 , …, A → A u_n} les règles récursives à gauche pour A, et soient
{A → v_1 , …, A → v_k | v_i ne commence pas par A} les autres règles en A.
Soit A' un nouveau non terminal associé à A :

- remplacer chaque règle A → A u_i par les règles

A' → u_i et A' → u_iA' pour i=1..n

- ajouter les règles : A → v_i A' pour i=1..k

B) Elimination des règles B →Au

Remarque : après la 1ère transformation (élimination des règles A→Au) on a nécessairement B >A (au sens de l'ordre défini)

- On remplace la règle B → Au par l'ensemble des règles

{ B → v u | A → v est une règle de la grammaire}.
Compte tenu de la remarque ci-dessus, v commence par un terminal

Grammaires linéaires
Une grammaire h.c. G est *linéaire* si le membre droit de chaque règle contient au plus un seul symbole non terminal.
Donc si toutes ses règles sont de la forme

- $A \rightarrow uBv$ où u, v $\in V_T^*$. Si $u=\varepsilon$ on dit que G est linéaire à droite. Si $v=\varepsilon$ on dit que G est linéaire à gauche
- $A \rightarrow u$ où $u \in V_T^*$

Décidabilité
Soit G = $< V_T, V_N, S, R >$ une grammaire hors contexte.
Pour tout $u \in V_T^*$ on peut décider si $u \in L_G(S)$ ou non.

Preuve

u = ε
L'ensemble U défini au §3 qui est la limite de la suite

- $U_0 = \{ A \in V_N \mid A \rightarrow \varepsilon$ est une règle de G $\}$
- $U_i = U_{i-1} \cup \{ A \in V_N \mid \exists\, v \in U^*_{i-1}$ tq $A \rightarrow v$ est une règle de G $\}$

est l'ensemble des non terminaux X tels que $X \rightarrow \varepsilon$
Si $S \in U$ la réponse est *oui* sinon la réponse est *non*.

u ≠ ε
Dans ce cas on peut supposer que G est réduite et propre.
Alors, si $|u| = k \Rightarrow [S \rightarrow^* u \Leftrightarrow S \rightarrow^n u$ et $n < 2k]$
Il suffit donc de construire toutes les dérivations de S de longueur inférieure à 2k et de regarder si u figure dans la liste des mots obtenus ou non.
Reste à montrer la propriété ci-dessus. G étant réduite et propre les membres droits des règles consistent en au moins 1 symbole terminal ou comportent plusieurs symboles. Il s'ensuit que chaque symbole du membre droit de chaque règle fait augmenter d'au moins 1 la longueur du mot produit. Donc :

- le nombre de règles terminales utilisées pour obtenir u est au plus k (chaque règle appliquée produisant au moins une des k lettres de u)
- les règles non terminales utilisées font augmenter la longueur du mot produit de 1 au moins. On en applique donc au plus k-1.

On applique donc au plus n = k + (k-1) =2k-1 règles pour obtenir u.

LEMME CARACTERISTIQUE
Ce lemme (dit « pumping lemma ») est utilisé pour montrer qu'un langage n'est pas hors contexte. Il dit que dans toute chaîne suffisamment longue d'une GHC, on peut trouver au plus deux courtes sous-chaînes voisines répétées *n* fois (*n* étant un entier quelconque), que l'on peut « épuiser » en parallèle, la chaîne restante faisant toujours partie du langage engendré par la GHC.

Enoncé
Soit G = $< V_T, V_N, S, R >$ une grammaire hors contexte. Il existe un entier K attaché à G tel que tout mot $z \in L(G)$ avec $|z| \geq K$, se factorise en $z = uvwxy$ avec :
(1) v et x ne sont pas simultanément égaux à ε : $|v|+|x| > 0$ ou $vx \neq \varepsilon$: x et v sont les parties à épuiser et au moins l'une des 2 n'est pas vide.
(2) $|vwx| \leq K$: la partie du « milieu » n'est pas trop longue

(2) $uv^nwx^ny \in L(G)$ pour tout $n \geq 0$: les 2 chaînes v et x peuvent être épuisées un nombre arbitraire de fois et le reste est toujours un mot du langage engendré par la GHC.

On admettra la démonstration. On peut prendre $K=M^{p+1}$ où :
$M = \max \{|w| \text{ tel que } A \rightarrow w \text{ est une règle}\}$ et $p = \#V_N$.

<u>Application</u>
On utilise ce lemme pour montrer sur un contre-exemple qu'un langage L n'est pas hors contexte. On peut voir les choses sous la forme suivante :

1. On suppose que L est HC et on prend le nombre K donné par le lemme
2. On choisit une chaîne z de L
3. On essaie de décomposer z en $uvwxy$ avec les contraintes $|vwx| \leq K$ et $vx \neq \varepsilon$
4. Si on trouve un entier n tel que uv^nwx^ny n'est pas dans L, alors le lemme est contredit et L n'est pas HC.

Exemple
$L=\{0^n\, 1^n\, 2^n \mid n > 0\}$. On veut montrer que L n'est pas HC.

1. On suppose que L est HC et on prend le nombre K donné par le lemme
2. On choisit $z = 0^K\, 1^K\, 2^K$
3. On essaie de décomposer z en $uvwxy$ avec $|vwx| \leq K$ et $vx \neq \varepsilon$. Alors on sait que vwx ne peut pas contenir à la fois des 0 et des 2 car le dernier 0 et le 1er 2 sont séparés par K+1 positions et $|vwx| \leq K$. Donc
   - Ou bien vwx ne contient pas de 2 et donc vwx ne contient que des 0 et des 1 (et au moins l'un d'eux) et comme Mais alors :
     - $|uvwxy|_0 = |uvwxy|_1 = |uvwxy|_2 = K$
     - $|vwx|_2 = 0$ et $|vwx|_0 + |vwx|_1 \geq 1$

     Dans ce cas tous les 2 se trouvent dans y : $|y|_2 = K$ et $|uvwx|_0 = |uvwx|_1 = K$ et puisque $vx \neq \varepsilon$, $|uw|_0 < K$ ou $|uw|_1 < K$. Donc uwy contient K fois 2, mais moins de K fois 1 ou moins de K fois 0. Donc $uwy \notin L$ ce qui contredit le lemme pour n=0. Donc L n'est pas HC.
   - Ou bien vwx ne contient pas de 0. Comme ci-dessus, on en déduit que uwy contient K fois 0 et moins de K fois 1 ou moins de K fois 2 et que L n'est pas HC.

Exemple
Cet exemple montre que les GHC ne peuvent pas établir une correspondance entre 2 chaînes de longueur arbitraire sur un alphabet de plus d'un symbole. Une conséquence est qu'on montre ainsi que les GHC ne permettent pas de « forcer » des contraintes sémantiques dans les langages de programmation du type « un identificateur doit être déclaré avant d'être utilisé ».

Soit $L = \{ww \mid w \in \{0,1\}^*\}$

1. On suppose que L est HC et on prend le nombre K donné par le lemme
2. On choisit $z = 0^K\, 1^K\, 0^K\, 1^K$ dans L
3. On essaie de décomposer z en $uvwxy$ avec $|vwx| \leq K$ et $vx \neq \varepsilon$. On remarque que $|uwy| \geq 3K$ car $|z| = 4K$. Donc si uwy est une chaine répétée, par exemple tt, alors $|t| \geq 3K/2$. Plusieurs cas sont possibles selon la position de vwx dans z :

- Si vwx est dans les K premiers '0'. En particulier, on peut avoir $vx = 0^p$ avec p>0. Alors uwy commence par $0^{K-p}1^K$. Puisque $|uwy| = 4K-p$, alors si $uwy = tt$, $|t| = K-p/2$. Donc t ne finit pas avant le premier bloc de '1' et donc t finit sur 0. Mais uwy finit sur 1 et ne peut donc pas être égal à tt.

- Si *vwx* est à cheval sur le premier bloc de ‘0’ et le premier bloc de ‘1’. En particulier, on peut avoir $vx = 0^p$ avec $p>0$ si $x = \varepsilon$. Alors de même que ci-dessus, *uwy* ne peut pas être de la forme *tt*. Si *vx* contient au moins un ‘1’, alors *t* qui est de longueur au moins $3K/2$ doit finir sur 1^K car *uwy* finit sur 1^K. Mais il n’y a pas de blocs de *K* ‘1’, sauf le dernier, donc *uwy* ne peut pas être de la forme *tt.*
- Si *vwx* est contenu dans le 1er bloc de ‘1’ alors *uxy* n’est pas dans L pour la même raison que ci-dessus.
- Si *vwx* est à cheval sur le 1er bloc de ‘1’ et le dernier bloc de ‘0’ : si *vx* ne contient pas de 0 alors le raisonnement est le même que si *vwx* est contenu dans le 1er bloc de ‘1’. Si *vx* contient au moins un 0 alors *uwy* commence par *K* ‘0’ et *t* aussi si *uwy* = *tt*. Mais alors il ne reste plus de bloc de *K* ‘0’ dans *uwy* pour le 2ème *t*. Donc, à nouveau, *uwy* n’est pas dans L.
- Pour les autres cas, *vwx* est dans la 2ème moitié de *z*, on fait des raisonnements symétriques à ceux où *vwx* est dans la 1ère moitié de *z*.

Donc *uwy* n’est jamais dans L qui n’est donc pas HC.

EQUIVALENCE AAP-GHC

Théorème

Si L est un langage hors contexte, alors il existe un automate à pile A tel que L= V(A).

Soit G la grammaire h.c. associée à L. On peut supposer que G est sous forme normale de Greibach. L ne contient pas ε.

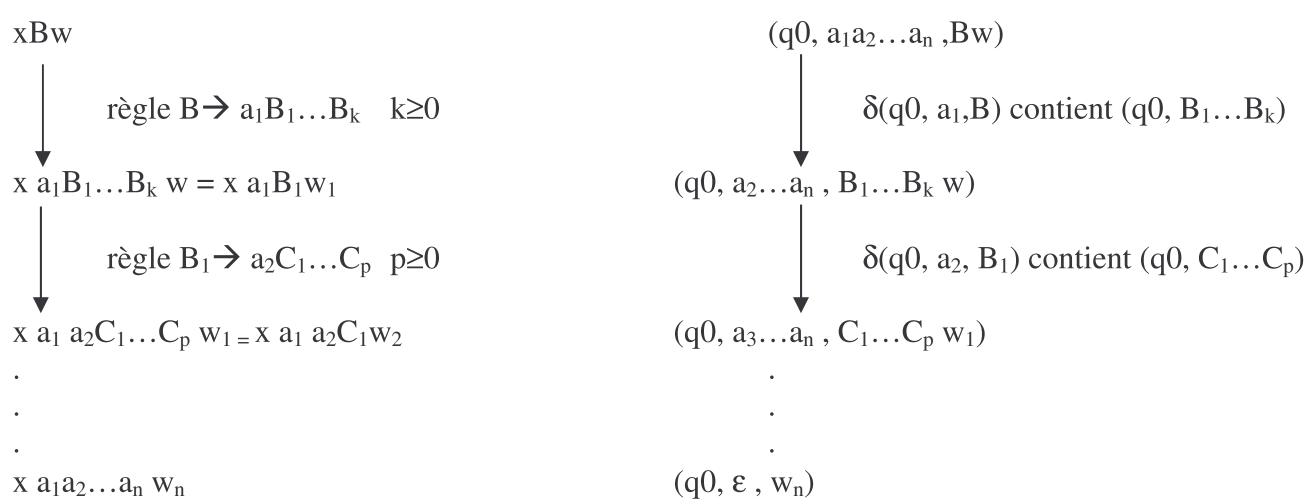
<u>Preuve par construction de A</u>

$G = <V_T, V_N, S, R>$

$A = <\{q0\}, V_T, V_N, \delta, S, \emptyset>$ où δ est défini par :

$\delta(q0,a,X) = \{(q0,u) \mid X \rightarrow au$ est une règle de G avec $a \in V_T$ et $u \in V_N^*\}$

$x \in V_T^*$, $B \in V_N$ $\quad$ $w \in V_N^*$



$w = w_n = x = \varepsilon$ et $B = S$

$S \rightarrow^* a_1a_2\ldots a_n$ $\quad$ $(q0, a_1\ldots a_n, S) \rightarrow^* (q0, \varepsilon, \varepsilon)$

$y = a_1a_2\ldots a_n$ est un mot du langage ssi $y = a_1a_2\ldots a_n$ est reconnu par pile vide

Exemple : L ={a^nb^n | n > 0}

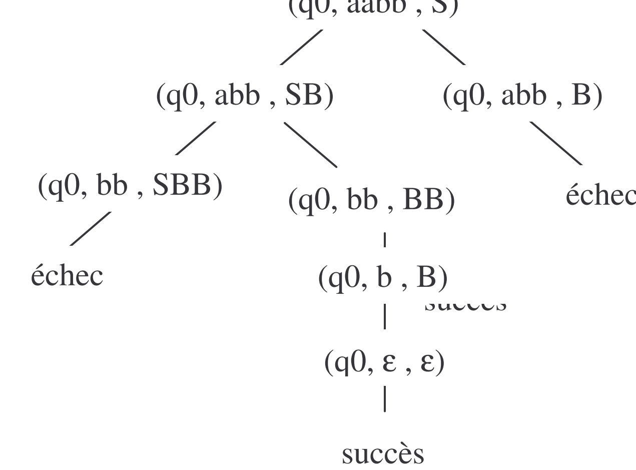
S→aSb S→ab

La forme normale de Greibach pour cette grammaire est :

S→aSB S→aB B→b

L'automate fonctionnant en mode pile vide est : A = < K, Σ , Γ , δ , q0 , Z , F > avec :

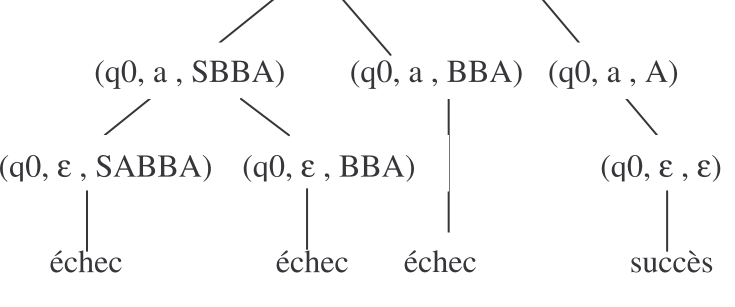
- q0 état initial
- Σ = {a,b}
- Γ = {S,B}
- S = symbole de fond de pile
- δ(q0, a, S) = {(q0, SB) ; (q0, B)}
- δ(q0, b, B) = {(q0, ε)}



Exemple : L = {WW~ | W ∈ {A,B} }

S→aSa
S→bSb
S→aa
S→bb

La forme normale de Greibach est
S→aSA
S→bSB
S→aA
S→bB
A→a
B→b



A = < K, Σ , Γ , δ , q0 , Z , F > avec :

- q0 état initial
- Σ = {a,b}
- Γ = {S,A,B}
- S = symbole de fond de pile
- δ(q0, a, S) = {(q0, SA) ; (q0, A)}
- δ(q0, b, S) = {(q0, SB) ; (q0, B)}
- δ(q0, a, A) = {(q0, ε)}
- δ(q0, b, B) = {(q0, ε)}

Théorème
S'il existe un automate à pile A tel que L= V(A) alors L est un langage hors contexte.

Preuve par construction de G
$A = < K, \Sigma , \Gamma , \delta , q0 , Z , \varnothing >$
$G = <V_T, V_N, S, R>$ est la grammaire h.c. définie par :

- $V_T = \Sigma$ *même alphabet terminal*
- $V_N = \{S\} \cup \{ [qAp] \mid p,q \in K, A \in \Gamma\}$ *un nouveau non terminal : l'axiome S et un nouveau non terminal noté [qAp] pour tout triplet (q,A,p) de KxΓxK*
- Règles

1. $S \rightarrow [q0Zq]$ *pour tout* $q \in K$

2. $[qAp] \rightarrow a\, [q_1B_1q_2]\, [q_2B_2q_3] \ldots [q_mB_mp]$
   pour tous $p, q_1,\ldots,q_m \in K$; $a \in \Sigma \cup \{\varepsilon\}$; $A, B_1, \ldots, B_m \in \Gamma$ *et* $m \neq 0$
   tels que $(q_1, B_1B_2\ldots B_m) \in \delta(q, a, A)$

3. $[qAp] \rightarrow a$
   pour tous $p, q \in K$; $a \in \Sigma \cup \{\varepsilon\}$ *et* $A \in \Gamma$ *tels que* $(p, \varepsilon) \in \delta(q, a, A)$

Pour montrer que L(G) = V(A) il suffit d'établir que : $[qAp] \rightarrow^* x$ ssi $(q, x, A) \rightarrow^*(p, \varepsilon, \varepsilon)$
Dès lors si $x \in L$, $S \rightarrow [q0Zq] \rightarrow^* x$ et donc $(q0, x, Z) \rightarrow^*(q, \varepsilon, \varepsilon)$ et $x \in V(A)$
De même si $x \in V(A)$, $(q0, x, Z) \rightarrow^*(q, \varepsilon, \varepsilon)$. Donc $S \rightarrow [q0Zq] \rightarrow^* x$ et $x \in L$

$(q, x, A) \rightarrow^*(p, \varepsilon, \varepsilon) \Rightarrow [qAp] \rightarrow^* x$
Par récurrence sur k tq $(q, x, A) \rightarrow^k (p, \varepsilon, \varepsilon)$

Si k=1, $x = \varepsilon$ ou $x = a \in V_T$. Alors $\delta (q, a, A)$ contient (p, ε) et donc :
$[qAp] \rightarrow a$ est une production de G (cf. 3.)

Supposons l'hypothèse vraie pour toute suite de transitions en moins de k étapes.

La première étape de la suite de transitions $(q, x, A) \rightarrow^k (p, \varepsilon, \varepsilon)$ est de la forme :
$(q, a, A) \rightarrow (q_1, \varepsilon, B_1\ldots B_m)$ où a est ε ou le 1er symbole de x : $x = a\, x_1\ldots x_m$
tels que : $(q_i, x_i, B_i) \rightarrow^*(q_{i+1}, \varepsilon, \varepsilon)$ en moins de k étapes.

Donc (hyp de réc.) $[q_iB_iq_{i+1}] \rightarrow x_i$.

Or $[qAp] \rightarrow a\, [q_1B_1q_2]\, [q_2B_3q_3]\ldots [q_mB_mp]$ est une production de G (cf. 2.)

et donc $[qAp] \rightarrow^* a\, x_1\ldots x_m = x$

$[qAp] \rightarrow^* x \Rightarrow (q, x, A) \rightarrow^*(p, \varepsilon, \varepsilon)$
récurrence sur k tq $[qAp] \rightarrow^k x$

Si k=1, $x = \varepsilon$ ou $x = a \in V_T$. Alors $(p, \varepsilon) \in \delta (q, x, A)$ et donc $(q, x, A) \rightarrow (p, \varepsilon, \varepsilon)$ (cf. 3.)

Supposons l'hypothèse vraie pour toute dérivation en moins de k étapes.
La première étape de la dérivation $[qAp] \rightarrow^k x$ est de la forme :

$[qAp] \rightarrow a\, x_1\ldots x_m$ où a est ε ou le 1er symbole de x : $x = a\, x_1\ldots x_m$

ce qui résulte de l'application de la règle :
$[qAp] \rightarrow a [q_1B_1q_2] [q_2B_2q_3] \dots [q_mB_mp]$ avec $(q_1, B_1\dots B_m) \in \delta(q, a, A)$ et :

- $[q_iB_iq_{i+1}] \rightarrow^* x_i$ pour $i = 1..m-1$ en moins de k étapes
- $[q_mB_mp] \rightarrow^* x_m$ en moins de k étapes

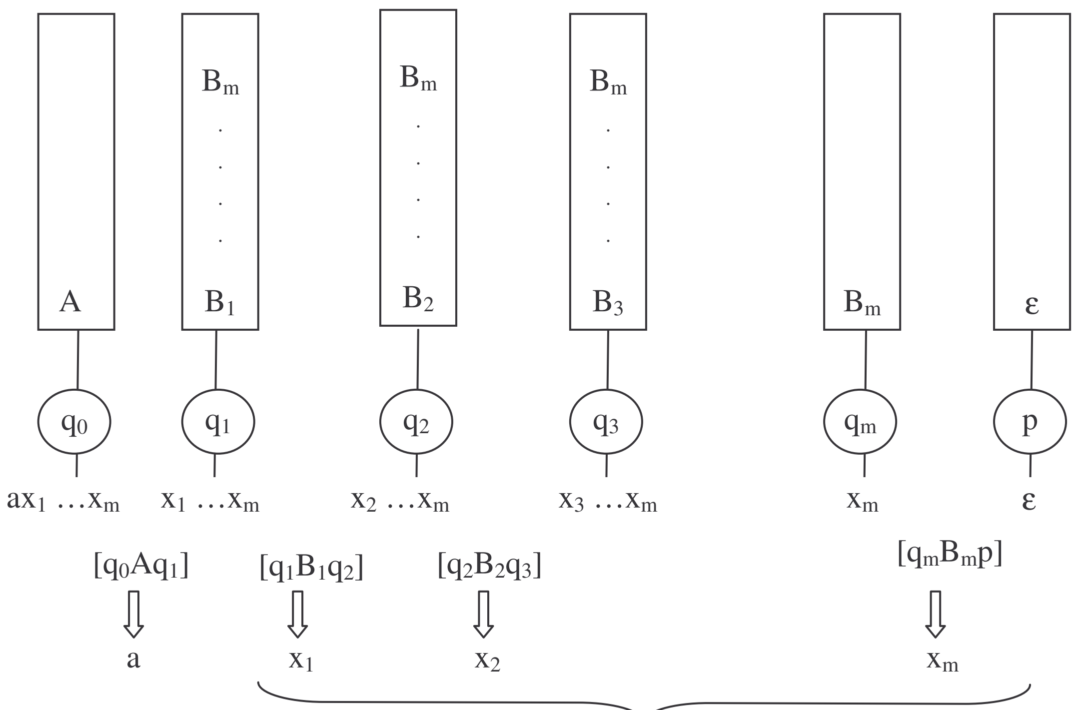
Donc (hyp. de réc.) $(q_i, x_i, B_i) \rightarrow^*(q_{i+1}, \varepsilon, \varepsilon)$ et $(q_m, x_m, B_m) \rightarrow^*(p, \varepsilon, \varepsilon)$

Comme de plus $(q, a, A) \rightarrow (q_1, \varepsilon, B_1\dots B_m)$ on a la suite de transitions :

$(q,a\, x_1\dots x_m,A)\rightarrow(q_1,x_1\dots x_m,B_1\dots B_m)\rightarrow(q_2, x_2\dots x_m, B_2\dots B_m)\dots\rightarrow(p, \varepsilon, \varepsilon)$

$x = ax_1 \dots x_m$ et $|x| = k$

Ceci peut être illustré par la figure ci-dessous :



Chapitre VII : ANALYSE DESCENDANTE DETERMINISTE – GRAMMAIRES ET LANGAGES LL(1)

DEFINITIONS

Soit G une grammaire, L le langage associé et w un mot construit sur le vocabulaire de G. On s'intéresse à savoir si oui ou non w est un mot de L, c'est à dire si S→*w.

Pour cela on dispose de 2 grandes stratégies d'analyse :

- *descendante* : partant de S on applique les règles de la grammaire de gauche à droite, on engendre tous les mots et on vérifie si w se trouve parmi les mots engendrés ou pas.
- *remontante* : partant de w, on applique les règles de la grammaire de droite à gauche en remontant jusqu'aux non-terminaux et on vérifie que S figure parmi les non terminaux produits.

Dans les 2 cas, on choisit les règles en fonction du mot w, et donc des terminaux qui le composent en lisant ces terminaux 1 par 1, de gauche à droite ou de droite à gauche.

Dans l'appellation LL(1), le 1er « L » signifie que la lecture se fait de gauche à droite, (Left).
Le 2ème « L » signifie que les dérivations portent sur le symbole non terminal le plus à gauche (Left).
Le « 1 » signifie que le choix de la règle à appliquer est déterministe dès que l'on connaît un seul symbole (donc le symbole courant) du mot à analyser

On définit de même les grammaires LL(k) : k symboles d'avance ou LR(k) : les dérivations se font par la droite (« R » : right).

Exemples
Soit la grammaire G1 :
(1) $S \rightarrow a$
(2) $S \rightarrow Ac$
(3) $A \rightarrow bAa$
(4) $A \rightarrow c$
Le choix de la règle à appliquer ne pose aucun problème :

non-terminal le plus à gauche	terminal courant	règle à appliquer
S	*a*	(1)
S	*b*	(2)
S	*c*	(2)
A	*b*	(3)
A	*a*	-
A	*b*	(3)
A	*c*	(4)

Si on ajoute la règle (5) $S \rightarrow bS$, la grammaire obtenue G2 n'est plus LL(1) :

non-terminal le plus à gauche	terminal courant	règle à appliquer
S	*b*	(2) & (5)

Soit la grammaire G3 :
(1) $S \rightarrow a$
(2) $S \rightarrow Ac$
(3) $A \rightarrow bAa$
(4) $A \rightarrow \varepsilon$

La présence de ε complique les choses. Pour reconnaître *bac* :
$S \rightarrow Ac \rightarrow bAac \rightarrow bac$

Le choix de la règle (4) repose sur la présence du a qui suit A ce qui est possible parce que la règle (4) a un membre droit vide.

Soit la grammaire G4 :
(1) $S \rightarrow Ac$
(2) $A \rightarrow \varepsilon$
(3) $A \rightarrow c$

Les 2 dérivations $S \rightarrow Ac \rightarrow cc$ et $S \rightarrow Ac \rightarrow c$ qui produisent toutes deux une chaine terminale commençant par *c* en utilisant des règles différentes montrent que la grammaire n'est pas LL(1).

CONDITIONS LL(1)

Dans ce paragraphe on définit un ensemble de critères qui permettent de décider si une grammaire est LL(1) ou pas.

On calcule pour chaque règle l'ensemble des symboles courants qui peuvent être produits par cette règle dans une dérivation partant de l'axiome. Cet ensemble est celui des *symboles directeurs* et est calculé comme suit.

Si $X \rightarrow w$ est une règle,

$\text{Dir}(X \rightarrow w)=\{y \in V_T \cup \{\$\} \mid \exists w_1,w_2,w_3 \text{ tq } S\$ \rightarrow^* w_1Xw_2 \rightarrow w_1ww_2 \rightarrow^* w_1yw_3\}$ où :
- \$ est un nouveau symbole $\notin V_T$
- $\text{Dir}(X \rightarrow w) \neq \emptyset$ pour toute règle $X \rightarrow w$
- $\varepsilon \notin \text{Dir}(X \rightarrow w)$ car $\varepsilon \notin V_T$
- lorsque *w* ne se dérive pas vers ε on peut simplifier la définition en :

$\text{Dir}(X \rightarrow w) = \{y \in V_T \mid \exists w_1 \text{ tq } X \rightarrow^* yw_1\}$

PROPRIETE FONDAMENTALE

Une grammaire G est LL(1) ssi elle vérifie la condition :
$\forall X \text{ tq } X \rightarrow w_1 \text{ et } X \rightarrow w_2 , w_1 \neq w_2 \Rightarrow \text{Dir}(X \rightarrow w_1) \cap \text{Dir}(X \rightarrow w_2)=\emptyset$

Exemple
Avec G2 :
- $S\$ \rightarrow Ac\$ \rightarrow bAac\$$ (règles 2 et 4)
- $S\$ \rightarrow bS\$$ (règle 3)

Donc $b \in \text{Dir}(S \rightarrow Ac) \cap \text{Dir}(S \rightarrow bS)$ et G2 n'est pas LL(1)

Dans G4, on peut construire :

- $S\$ \rightarrow Ac\$ \rightarrow c\$$ (règle 2)
- $S\$ \rightarrow Ac\$ \rightarrow cc\$$ (règle 3)

Donc $c \in \text{Dir}(A \rightarrow c) \cap \text{Dir}(A \rightarrow \varepsilon)$ et G4 n'est pas LL(1)

CALCUL DES ENSEMBLES DIRECTEURS

On pose $V'_T = V_T \cup \{\$\}$ et $V = V'_T \cup V_N$

Soit Premier : $V^* \rightarrow \{\text{parties de } V'_T\}$ et Suivant : $V_N \rightarrow \{\text{parties de } V'_T\}$

- Premier(w) = $\{y \in V'_T \mid \exists w_1 \in V_T^* \text{ et } w \rightarrow^* yw_1\}$. Ce sont les 1ers symboles terminaux apparaissant à gauche d'une dérivation de w.
- Suivant(X) = $\{y \in V'_T \mid \exists w_1\ w_2 \in V_T^* \text{ et } S\$ \rightarrow^* w_1Xyw_2\}$. Ce sont les 1ers symboles terminaux apparaissant après X quand on dérive X à partir de S

On a alors :

Premier	- Premier(ε) = $\emptyset$ - Premier(x) = $\{x\}$ pour $x \in V'_T$ - Premier(X) = $\bigcup_{X \rightarrow w_i}$ Premier(w_i) pour $X \in V_N$ - Premier(Xw) = Premier(X) $\cup$ (si $X \rightarrow^* \varepsilon$ alors Premier(w) sinon $\emptyset$) pour $X \in V$ et $w \in V^*$
Suivant	- Suivant(S) = $\bigcup_{Y \rightarrow w_1 S w_2}$ Premier(w_2) $\bigcup_{w_2 \rightarrow^* \varepsilon}$ Suivant(Y) $\cup \{\$\}$ pour $Y \in V_N$, $w_1\, w_2 \in V^*$ - Suivant(X) = $\bigcup_{Y \rightarrow w_1 X w_2}$ Premier(w_2) $\bigcup_{w_2 \rightarrow^* \varepsilon}$ Suivant(Y) si $X \neq S$ pour $X, Y \in V_N$, $w_1\, w_2 \in V^*$
Directeur	- Dir($X \rightarrow w$) = Premier(w) $\cup$ (si $w \rightarrow^* \varepsilon$ alors Suivant(X) sinon $\emptyset$) pourvu que la grammaire ne contienne pas de symbole inaccessible.

Exemple

$A \rightarrow AB$ $\quad A \rightarrow \varepsilon \quad B \rightarrow a$

Premier(A) = Premier(A) $\cup$ Premier(B) et Premier(B) = $\{a\}$

La 1ère équation a pour plus petite solution Premier(A) = Premier(B) = $\{a\}$. En effet, d'une façon générale, $X = X \cup B$ a pour (plus petite) solution $X = B$.

C'est toujours cette plus petite solution qu'on retient ce qui garantit qu'on obtient toujours des éléments accessibles par dérivation.

Exemple

$S \rightarrow AB \quad B \rightarrow aAB \quad B \rightarrow \varepsilon \quad A \rightarrow CD \quad D \rightarrow bCD \quad D \rightarrow \varepsilon \quad C \rightarrow dSd \quad C \rightarrow c$

Dir($S \rightarrow AB$) = Premier(A)
Dir($B \rightarrow aAB$) = $\{a\}$
Dir ($B \rightarrow \varepsilon$) = Suivant($B$)
Dir ($A \rightarrow CD$) = Premier(C)
Dir ($D \rightarrow bCD$) = $\{b\}$
Dir ($D \rightarrow \varepsilon$) = Suivant($D$)
Dir ($C \rightarrow dSd$) = $\{d\}$
Dir ($C \rightarrow c$) = $\{c\}$

Suivant(B) = Suivant(B) $\cup$ Suivant(S)
Suivant(S) = $\{d,\$\}$
Suivant(D) = Suivant(D) $\cup$ Suivant(A)
Suivant(A) = Premier(B) $\cup$ Suivant(B) $\cup$ Suivant(S)

D'autre part :
Premier(A) = Premier(C) = $\{d,c\}$
Premier(B) = $\{a\}$

Et :
Suivant(B) = $\{d,\$\}$
Suivant(S) = $\{d,\$\}$
Suivant(D) = Suivant(A) = Premier(B) $\cup$ $\{d,\$\}$ = $\{a, d, \$\}$

Donc :
Dir($S \rightarrow AB$) = $\{d,c\}$
Dir($B \rightarrow aAB$) = $\{a\}$
Dir ($B \rightarrow \varepsilon$) = $\{d,\$\}$
Dir ($A \rightarrow CD$) = $\{d,c\}$
Dir ($D \rightarrow bCD$) = $\{b\}$
Dir ($D \rightarrow \varepsilon$) = $\{a,d,\$\}$
Dir ($C \rightarrow dSd$) = $\{d\}$
Dir ($C \rightarrow c$) = $\{c\}$

} pas d'ambiguité sur le choix des règles pour *B, C,* et *D* ni pour *A* et *S* auxquels une seule règle est attachée
La grammaire est LL(1)

RESULTATS

On étend les propriétés des grammaires aux langages.
On dit donc qu'un langage est LL(1) ssi il existe une grammaire LL(1) qui l'engendre. Cependant, on ne sait pas décider si un langage est LL(1) ou non, car il n'existe pas d'algorithme prenant en entrée une grammaire hors contexte et calculant l'existence d'une grammaire LL(1) équivalente.

On a cependant les résultats suivants :
- une grammaire ambiguë n'est pas LL(1)
- une grammaire récursive à gauche n'est pas LL(1)
- un langage régulier est toujours LL(1)

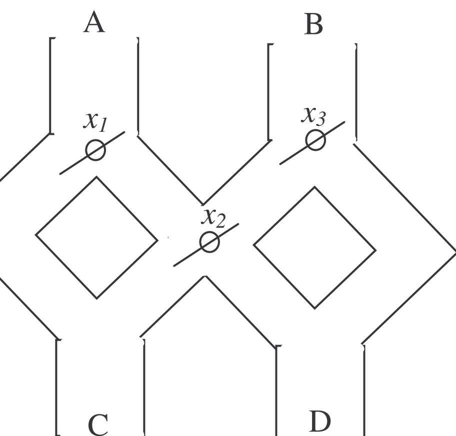
A noter que ce 3ème résultat n'apporte rien de plus à ce qu'on sait, puisque pour un langage régulier on sait déjà construire l'automate d'états finis déterministe associé.

EXERCICES

EXERCICES – CHAPITRE I : AUTOMATES ET LANGAGES

Exercice 1

Une bille est introduite dans les entrées A ou B du labyrinthe ci-contre. Selon la position des portes x_1, x_2 et x_3, la bille se dirige vers la droite ou vers la gauche. Chaque fois que la bille rencontre une porte, celle ci bascule après la passage de la bille.



Modéliser ce jeu par un automate fini dans lequel :

- les états initiaux sont A et B selon que la bille est introduite en A ou en B.
- l'acceptation correspond à la sortie en D.
- le rejet à la sortie en C.

Exercice 2

Soit l'alphabet X ={a, b} et l'homomorphisme $\varphi : (X^* , .) \rightarrow (\mathbb{Z} , +)$ tel que $\varphi(a) = 1$ et $\varphi(b) = -1$.
Pour $u, v \in X^*$ on dit que v est un facteur gauche de u et on écrit $v<u$ si $\exists\, x \in X^*$ tel que $u=vx$.
Soit $A = \{u \in X^* \mid \varphi(u) = -1\}$ et $L = \{u \in A \mid \forall\, v<u, v \neq u$, on a : $\varphi(v) \geq 0\}$

1. Comment interprétez-vous φ et que représentent A et L ?
2. Vérifier que le seul mot de L qui commence par b est b lui même.
3. En posant L' = L – {b}, montrer que tout mot u de L' s'écrit $u = axy$ avec x et y dans L.
4. Montrer que tout mot u de A se décompose de manière unique en $u = xy$ tels que $yx \in$ L.
5. Montrer que $L^n = \{u \in X^* \mid \varphi(u) = -n$ et $\forall\, v<u, v \neq u$, on a : $\varphi(v) > -n\}$

EXERCICES – CHAPITRE II : AUTOMATES D'ETATS FINIS

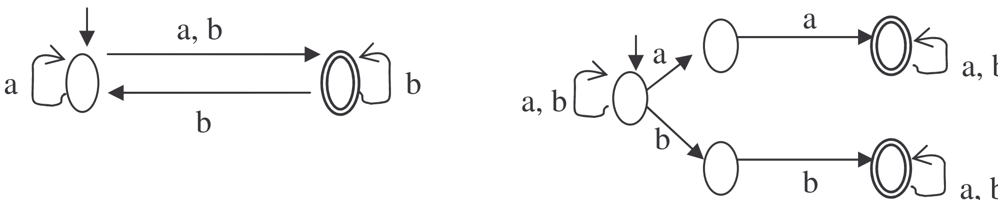
Exercice 1

Donner les AEF reconnaissant les langages suivants sur le vocabulaire V={a , b}

a) L1 = V*
b) L2 = {*w* | *w* contient 2 « *a* » consécutifs}
c) L3 = { $a^n\ b^m$ | *n, m* >0 }
d) L4 = {*w* | longueur de *w* = 3*k*+1, *k*≥0}
e) L5 = { *w* | *w* ne contient pas 2 « *a* » consécutifs }
f) L6 = { $a^n\ b^m$ | *n+m* pair }
g) L7 = { $a^n\ b^m$ | *n+m* impair }

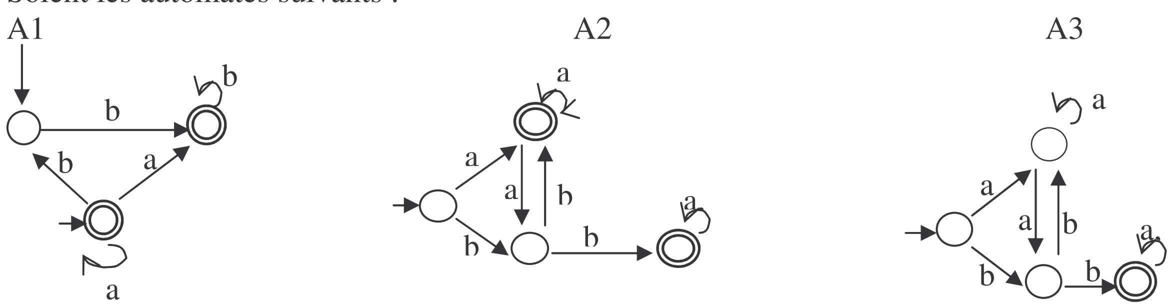
Exercice 2

Dire en quoi les AEF suivants sont non déterministes et construire les AEF déterministes équivalents.



Exercice 3

Soient les automates suivants :



Si L1=L(A1) ; L2=L(A2) et L3 = L(A3) déterminer les automates reconnaissant respectivement L1.L2 et L3*

Exercice 4

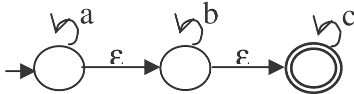
Soit l'AEF avec ε-transitions défini par la table suivante :

	ε	a	b	c
→p	∅	{p}	{q}	{r}
q	{p}	{q}	{r}	∅
*r	{q}	{r}	∅	{p}

1) Déterminer l'ε-fermeture de chaque état
2) Donner toutes les chaînes de longueur 3 ou moins acceptées par cet automate
3) Donner l'automate déterministe équivalent

Exercice 5

Eliminer les ε-transitions de l'AEF :



EXERCICES – CHAPITRE III : SYSTEMES DE RE-ECRITURE ET GRAMMAIRES

Exercice 1
V={a,b,c}. Donner une grammaire engendrant $\{a^n b^n c^p \mid n,p > 0\}$

Exercice 2
V={a,b}. Donner une grammaire engendrant les palindromes pairs : aa, abba, aabbaa, …

Exercice 3
V={a,b}. Donner une grammaire engendrant les chaines ne contenant pas 2 « b » consécutifs

Exercice 4
V={a,b ,c}. Donner une grammaire engendrant $\{a^m b c^p \mid m \geq p\}$

Exercice 5
V={a,b,c}. Donner une grammaire engendrant $\{a^p b^q c^r \mid p+q \geq r\}$

Exercice 6
V={a,b c}. Donner une grammaire engendrant $\{a^n b^n c^n \mid n>0\}$

Exercice 7
V={a,b}. Donner une grammaire engendrant les chaines contenant autant de « a » que de « b »

Exercice 8
V={a, +, = }. Donner une grammaire engendrant les chaines représentant une addition « correcte » : aa+aaa=aaaaa

Exercice 9
V={/, _ , \}. Quel est le langage engendré par la grammaire :

S→UAV S→UV A→VSU A→VU U→_/ V→_

Exercice 10
V={a,b,c}. Soit G la grammaire suivante où S est l'axiome:

S→aSA	S→bSB	S→c
cA→ca	cB→cb	aA→Aa
aB→Ba	bA→Ab	bB→Bb

1) Quel est le type de cette grammaire
2) Donner une dérivation de w=abcab
3) Montrer que L(G') = {w c w| w ∈ {a,b}*}

Exercice 11
V={# , a}. Soit G la grammaire suivante où S est l'axiome:

S→#a# #a→#B Ba→aaB B#→aa#

Quel est le langage engendré ?

EXERCICES – CHAPITRE IV : GRAMMAIRES ET LANGAGES REGULIERS

Exercice 1

Construire un automate de vocabulaire {0 , 1} reconnaissant le langage défini par l'expression régulière ((10)*1*)*.

Exercice 2

Soit la grammaire :

S→ aA S →bB A→ bB A→ aC A→a
B→aA B→bC B→b C→aC C→bC C→a C→b

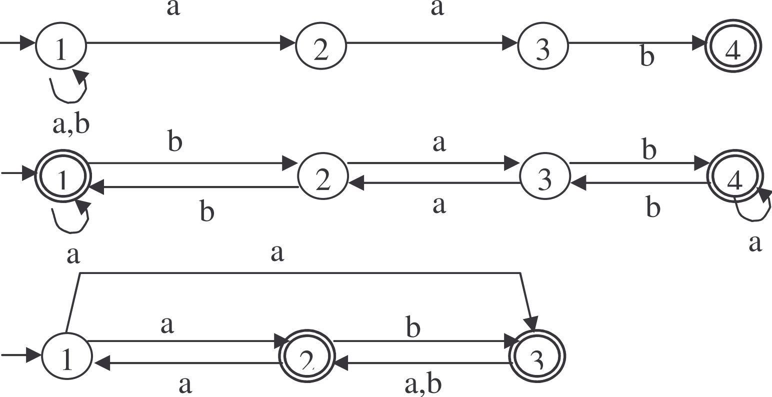
Construire l'AEF associé et écrire l'expression régulière du langage engendré

Exercice 3

V={a,b}. Montrer que la grammaire :
S→abbaS S→aA S→ab A→bbaB B→aS
est équivalente à une grammaire régulière

Exercice 4

Ecrire les expressions régulières des langages reconnus par chacun des automates suivants



Exercice 5

Prouver (ou démentir) chacune des identités suivantes sur les expressions régulières :

1) (RS+R)*R = R(SR+R)*
2) (RS+R)*RS = (RR*S)*
3) ((R+S)*S = (R*S)*
4) S(RS+S)*R = RR*S(RR*S)*

EXERCICES – CHAPITRE V : AUTOMATES A PILE

Dans ces exercices, on demande de construire les AAP reconnaissant les langages définis sur les vocabulaires indiqués

Exercice 1

$\{w = a^n \, b^n \mid n>0\}$

Exercice 2

$\{w \in \{a,b\}^* \mid w \text{ contient autant de a que de b}\}$

Exercice 3

$\{w = a^n \, b^m \mid n \leq m \leq 2n\}$

Exercice 4

$\{w = b^n \, a \, (ba)^* \, c^n \mid n \geq 1\}$

Exercice 5

L'ensemble des chaînes de 0 et de 1 dans lesquelles aucun préfixe n'a plus de 1 que de 0

Exercice 6

L'ensemble des chaînes de 0 et de 1 avec deux fois plus de 0 que de 1

Exercice 7

$\{w = a^i \, b^j \, c^k \mid i = j \text{ ou } j = k\}$

EXERCICES – CHAPITRE VI : GRAMMAIRES HORS-CONTEXTE

Exercice 1

Soit la grammaire G d'axiome S sur le vocabulaire {a,b} suivante :

S→aSc S→aS S→T

T→bTc T→bT T→ε

1) Montrer que L(G) = $\{a^p\ b^q\ c^r \mid p+q \geq r \geq 0\}$
2) Vérifiez que G est ambiguë et proposer une grammaire équivalente non ambiguë

Exercice 2

Soit la grammaire G d'axiome S sur le vocabulaire {0,1} suivante :

S→0S11 S→0S1 S→ε

1) Quel est le langage engendré par G ?
2) Montrer que le nombre de dérivations de la chaine $0^n\ 1^n$ avec $n \leq m \leq 2n$ est $C_n^{2n-m} = C_n^{m-n}$

Exercice 3

Soit L= {w ∈ {a,b}* | w contient autant de a que de b} et les 3 grammaires :

- G1 : S→aSb S→bSa S→SS S→ε
- G2 : S→aSbS S→bSaS S→ε
- G3 : S→ aB S→bA S→ε A→aS A→bAA B→bS B→aBB

1) Montrer que ces 3 grammaires engendrent L
2) Montrer qu'elles sont toutes 3 ambiguës et en trouver une qui ne le soit pas.

Exercice 4

Soit la grammaire : S→SS S→a

1) Quel est le langage engendré ?
2) Montrer que la grammaire est ambiguë
3) Soit X_n le nombre de dérivations gauches de a^n. Donner une relation de récurrence satisfaite par X_n.
4) Prouver que $X_{n+1} = (1/n+1)\ C_{2n}^n$

Exercice 5

Déterminer l'ensemble des symboles productifs de la grammaire :

A→B B→C C→D D→E D→a E→EA

Exercice 6

Soit la grammaire : S→A S→a A→CD C→c D→A E→B

Eliminer les symboles inaccessibles, puis les symboles non productifs et réduire la grammaire obtenue

Exercice 7

Soit la grammaire :

S→ABC A→BB A→ε B→CC B→a

C→AA C→b

Trouver une grammaire équivalente sans ε-règle

EXERCICES – CHAPITRE VII : LANGAGES LL(1)

Exercice 1

Les grammaires suivantes sont-elles LL(1) ?

1) $S \rightarrow Abc$ $S \rightarrow aAbc$ $A \rightarrow b$ $A \rightarrow c$ $A \rightarrow \varepsilon$

2) $E \rightarrow E+T$ $E \rightarrow E - T$ $E \rightarrow T$ $T \rightarrow T*F$ $T \rightarrow T / F$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow a$

3) $E \rightarrow TD$ $T \rightarrow FV$ $F \rightarrow (E)$ $F \rightarrow a$
$D \rightarrow UD$ $D \rightarrow \varepsilon$ $U \rightarrow + T$ $U \rightarrow - T$
$V \rightarrow GV$ $V \rightarrow \varepsilon$ $G \rightarrow *F$ $G \rightarrow / F$

Exercice 2

Donner les grammaires LL(1) engendrant les langages suivants sur le vocabulaire {a, b, c, d}

1) $\{w \in \{a,b\}^* \mid w$ contient autant de a que de b$\}$.

2) $\{w = xcycx^{\sim} \mid x , y \in \{a,b\}^*$ et $x^{\sim}$ est le miroir de x$\}$

3) $\{w = xdydy^{\sim} \mid x , y \in \{a,b\}^*$ et $y^{\sim}$ est le miroir de y$\}$