Université de Jijel
Département d'informatique
Mr.HEMIOUD Mourad
hemourad@yahoo.fr

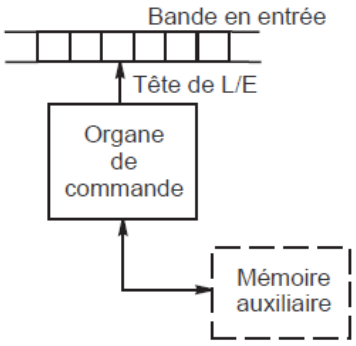
.1 Introduction

Les ***grammaires*** représentent un moyen qui permet de *décrire* un langage d'une manière inductive. Elles montrent comment les mots du langage sont construits.

Un ***automate*** est une machine abstraite qui permet de lire un mot et de répondre à la question : "un mot ***w*** appartient-il à un langage ***L*** ?" par ***oui*** ou ***non***. Aucune garantie n'est cependant apportée concernant le temps de reconnaissance ou même la possibilité de le faire.

Un automate est composé de :

- Une bande en entrée finie ou infinie sur laquelle sera inscrit le mot à lire ;
- Un organe de commande qui permet de gérer un ensemble fini de pas d'exécution ;
- Eventuellement, une mémoire auxiliaire de stockage.



Formellement, un automate contient au minimum :

- Un alphabet pour les mots en en entrée noté ***A*** ;
- Un ensemble non vide d'états noté **Q**;
- Un état initial noté $q_0 \in Q$;
- Un ensemble non vide d'états finaux $q_f \in Q$;
- Une fonction de transition (permettant de changer d'état) notée **δ** (est une fonction totale de Q x *A* dans Q, $\delta(p,a)=q \ / p,q \in Q$)

.2 Configuration d'un automate

Le fonctionnement d'un automate sur un mot se fait à travers un ensemble de configurations.

Définition : On appelle *configuration d'un automate* en fonctionnement les valeurs de ses différents composants, à savoir la position de la tête L/E, l'état de l'automate et éventuellement le contenu de la mémoire auxiliaire (lorsqu'elle existe).

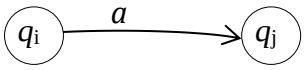
Il existe deux configurations spéciales :

1. La configuration ***initiale*** est celle qui correspond à l'état initial q_0 et où la tête de L/E est positionnée sur le premier symbole du mot à lire.
2. Une configuration ***finale*** est celle qui correspond à un des états finaux q_f et où le mot a été entièrement lu.

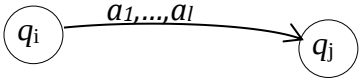
2.3 Représentation graphique d'un automate

Un automate fini correspond à un graphe orienté
Chaque état $\boldsymbol{q_i}$ de l'automate est schématisé par un rond (ou sommet).

Si la transition $\boldsymbol{\delta(q_i, a) = q_j}$ est définie, alors on raccorde le sommet $\boldsymbol{q_i}$ au sommet $\boldsymbol{q_j}$ par un arc décoré par le symbole $\boldsymbol{a}$



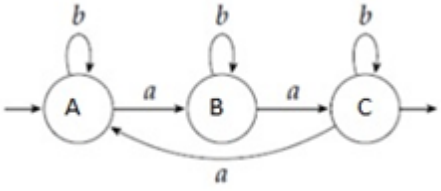
Lorsqu'il y a plusieurs symboles $\boldsymbol{a_1,\ldots,a_l}$ tel que $\delta(q_i, a_l) = q_j$



Etat initiale (q_0):

Etats finaux (q_f)

Exemple : Représentation graphique de l'automate
X=(A={a,b},Q={A,B,C},q_0=A,q_F={C},δ) tel que :
δ (A, *a*)=B
δ (B, *a*)=C
δ (A, *b*)=A
δ (B, *b*)=B
δ (C, *b*)=C



2.4 Mot reconnu par un automate

On dit qu'un mot **est reconnu par un automate** si, à partir d'une configuration initiale, on arrive à une configuration finale à travers une *succession de configurations* intermédiaires.

Les ***configurations successives*** d'un automate sont déterminées à partir de δ et nous notons quand on parle de configuration δ « ⊢ » et ainsi δ* devient « ⊢*»

Un mot *w* est ***reconnu par l'automate*** A s'il existe une *configuration successive* Configuration-initiale (*w*) ⊢* configuration-final(*w*) $(q_0,w) \vdash^* (q_f,\varepsilon)$

La relation ⊢ permet de formaliser la notion d'étape élémentaire de calcul d'un automate. Ainsi on écrira, pour a dans A et v dans A* : $(q, \underline{a}v) \vdash (\delta(q, a); v)$

2.5 Langage reconnu par un automate

On dit qu'un ***langage est reconnu par un automate*** X lorsque tous les mots de ce langage sont reconnus par l'automate on note L(X)

L(X)={$w \in$A* / Configuration-initiale (*w*) ⊢* configuration-final(*w*)}

L(X)={ $w \in$A* / $(q_0, w) \vdash^* (q, \varepsilon)$, avec q$\in Q_F$ }

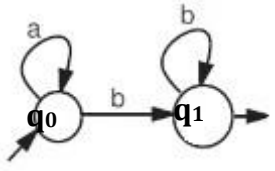
Cette notation met en évidence l'automate comme une machine permettant de reconnaître des mots : tout parcours partant de q_0 permet de « consommer » un à un les symboles du mot à reconnaître ; ce processus stoppe lorsque le mot est entièrement consommé : si l'état ainsi atteint est final, alors le mot appartient au langage reconnu par l'automate.

Exemple 1 : L'automate qui reconnaît les mots de la forme $a^n b^m (n > 0, m > 0)$ est le suivant : ({a, b}, { q_0, q_1}, q_0, { q_1}, δ) tel que δ est donnée par :

1) $\delta(q_0,a)=q_0$
2) $\delta(q_0,b)=q_1$
3) $\delta(q_1,b)=q_1$

ou par la table :

Etat	*a*	*b*
q_0	q_0	q_1
q_1	-	q_1



Essayons à présent d'analyser certains mots par cet automate

- **Mot1=aab**

  Configuration initiale : (q_0, **a**ab) (la tête de L/E est positionnée sur le premier symbole du Mot1.
  Configuration final :(q_1,ε)(l'états finale q_1 et le mot a été entièrement lu)

  $(q_0, \underline{\mathbf{a}}ab) \vdash (q_0, \underline{\mathbf{a}}b) \vdash (q_0, \underline{\mathbf{b}}) \vdash (q_1, \boldsymbol{\varepsilon})$

Voilà il existe une configuration successive
Configuration-initiale (*Mot1*) ⊢* configuration-final(*Mot1*)

$$(q_0, \underline{\mathbf{a}}ab) \vdash^* (q_1, \boldsymbol{\varepsilon})$$

Donc le mot Mot1=aab est reconnu par l'automate

- **Mot2=aa**

$(q_0, \underline{\mathbf{a}}a) \vdash (q_0, \underline{\mathbf{a}}) \vdash (q_0, \underline{\boldsymbol{\varepsilon}})$

Donc Mot2=**aa** n'est pas reconnu par l'automate (le mot a été entièrement lu et l'état q_0 n'est pas finale)

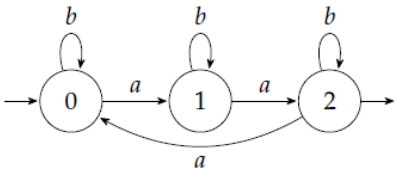
- **Mot3=aabba**

$(q_0, \underline{\mathbf{a}}abba) \vdash (q_0, \underline{\mathbf{a}}bba) \vdash (q_0, \underline{\mathbf{b}}ba) \vdash (q_1, \underline{\mathbf{b}}a) \vdash (q_1, \underline{\mathbf{a}})$
Donc Mot2=**aabba** n'est pas reconnu par l'automate

Exemple 2 :

La fonction de transition correspondant à ce graphe s'exprime matriciellement par :

δ	a	b
0	1	0
1	2	1
2	0	2



- Donnez 3 mots qui sont reconnu par cet automate (justifier votre réponse par la configuration successive)

2.6 Langage reconnaissable

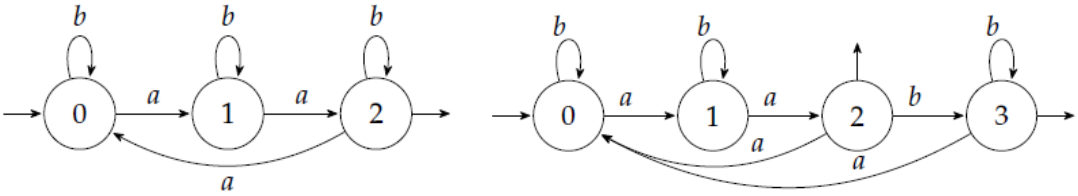
Définition (Langage reconnaissable)**.** Un langage est reconnaissable s'il existe un automate fini qui le reconnaît.

2.7 Automates équivalents

Définition (Automates équivalents) :

Deux automates finis X_1 et X_2 sont équivalents si et seulement s'ils reconnaissent le même langage ($L(X_1)$=$L(X_2)$)

Exemple



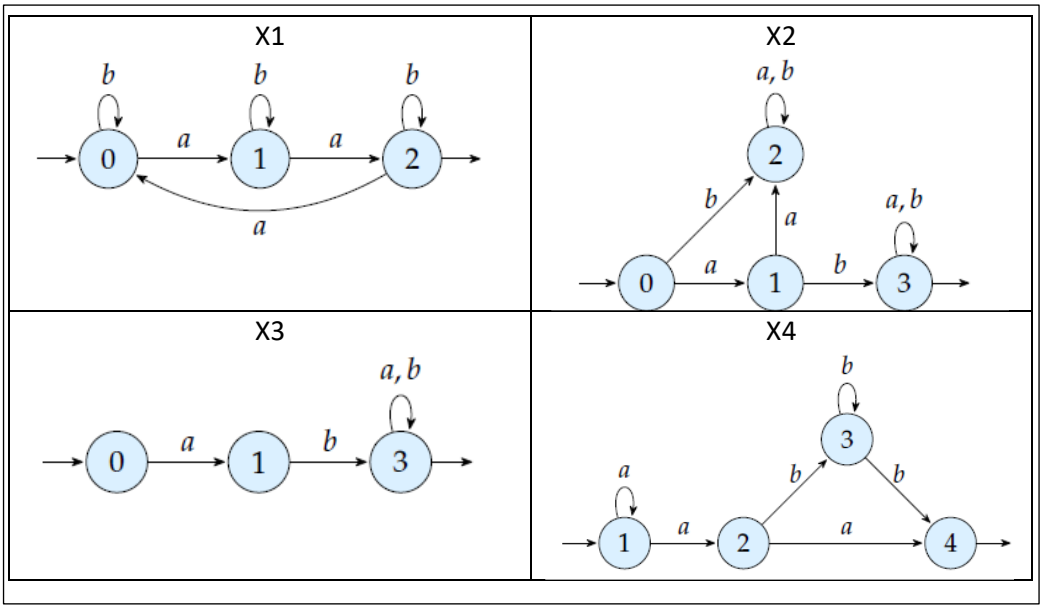
Les deux automates reconnaissent le langage de tous les mots qui contiennent un nombre de ***a*** congru à 2 modulo 3.

2.8 Automate fini déterministe AFD

Définition : Un AEF (A,Q, q_0, Q_F, δ) est dit **déterministe** si les deux conditions sont vérifiées :

- $\forall q_i \in Q, \forall a \in X$, il existe au plus un état q_j tel que $\delta(q_i, a) = q_j$;
- L'automate ne comporte pas de ε-transitions.

Exemples



- X1, X2 et X3 sont des automates déterministes
- X3 non déterministe

Algorithme de reconnaissance d'un mot par un AFD

On dérive un **algorithme permettant** de tester si un mot appartient au langage reconnu par un automate fini **déterministe**.

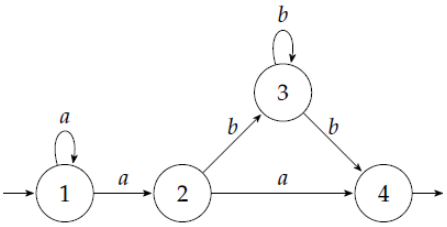
La **complexité** de cet algorithme découle de l'observation que chaque étape de calcul correspond à une application de la fonction δ(), qui elle-même se réduit à la lecture d'une case d'un tableau et une affectation, deux opérations qui s'effectuent en temps constant. La reconnaissance d'un mot ***w*** se calcule en exactement ***|w|*** étapes.

Algorithme : – Reconnaissance par un AFD

// $w = w_1 \ldots w_n$ est le mot à reconnaître.
// X = (A;Q; q_0; Q_F; δ) est le AFD.
q := q_0;
for i := 1 **to** n **do**
q := δ(q; w_i)
if q ∈ Q_F **then return** true **else return** false ;

AEF non-déterministe (déterminisation)

L'exemple suivant illustre ces notions.
Un automate non-déterministe



Deux transitions sortantes de 1 sont étiquetées par a : δ(1, a) = {1, 2}. ***aa*** donne lieu à un calcl réussi passant successivement par 1, 2 et 4, qui est final ((1,aa) ⊢ (2,a) ⊢ (4,ε)); ***aa*** donne aussi lieu à un calcul (1,aa) ⊢ (1,a) ⊢(2,ε) ou ((1,aa) ⊢ (1,a) ⊢ (1,ε)) qui n'est pas un calcul réussi.

Le langage reconnu par un automate non-déterministe

Pour qu'un mot appartienne au langage reconnu par l'automate, il suffit qu'il existe, parmi tous les calculs possibles, un calcul réussi,

La reconnaissance n'échoue donc que si tous les calculs aboutissent à une des situations d'échec.

Ceci implique que pour calculer l'appartenance d'un mot à un langage, il faut, en principe, examiner successivement tous les chemins possibles, et donc éventuellement revenir en arrière dans l'exploration des parcours de l'automate lorsque l'on rencontre une impasse.

Dans ce nouveau modèle, le problème de la reconnaissance d'un mot reste donc polynomial en la longueur du mot en entrée.

Note : La généralisation du modèle d'automate fini liée à l'introduction de transitions non-déterministes est, du point de vue des langages reconnus, sans effet : tout langage reconnu par un automate fini non-déterministe est aussi reconnu par un automate déterministe.

Théorème (Déterminisation)**.** Pour tout AFN ***X***, on peut construire un AFD ***X'*** équivalent à ***X***. De plus, si ***X*** a *n* états, alors ***X'*** a au plus 2^n états.

Déterminisation d'un AEF sans ε-transition

En réalité, l'algorithme de déterminisation d'un AEF est général, c'est-à-dire qu'il fonctionne dans tous les cas (qu'il y ait des ε-transitions ou non). Cependant, il est plus facile de considérer cet algorithme sans les ε-transitions. Dans cette section, on suppose que l'on a un AEF A ne comportant aucune ε-transition.

Algorithme : Déterminiser un AEF sans les ε-transitions

Principe : considérer des ensembles d'états plutôt que des états (dans l'algorithme suivant, chaque ensemble d'états représente un état du futur automate).

1- Partir de l'état initial $E^{(0)} = \{q_0\}$ (c'est l'état initial du nouvel automate) ;
2- Construire $E^{(1)}$ l'ensemble des états obtenus à partir de $E^{(0)}$ par la transition a :

$$E^{(1)} = \bigcup_{q' \in E(0)} \delta(q', a)$$

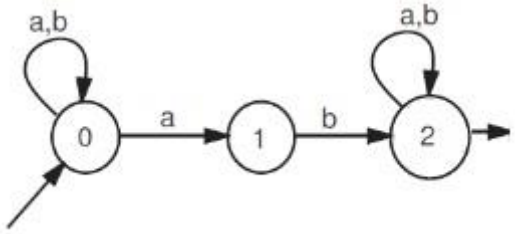
3- Recommencer l'étape 2 pour toutes les transitions possibles et pour chaque nouvel ensemble $E^{(i)}$;

$$E^{(i)} = \bigcup_{q' \in E(i-1)} \delta(q', a)$$

4- Tous les ensembles contenant au moins un état final du premier automate deviennent finaux ;
5- Renuméroter les états en tant qu'états simples.

Pour illustrer cet algorithme, nous allons l'appliquer à l'automate suivant
La table suivante illustre les étapes d'application de l'algorithme (les états en gras sont des états finaux) :

L'automate (non-déterministe) des mots contenant le facteur ab

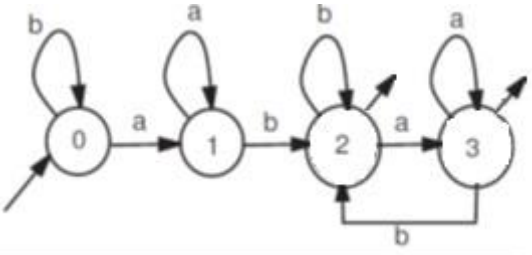


État	a	b
0	0,1	0
0,1	0,1	0,2
0,2	0,1,2	0,2
0,1,2	0,1,2	0.2

$\Longrightarrow$

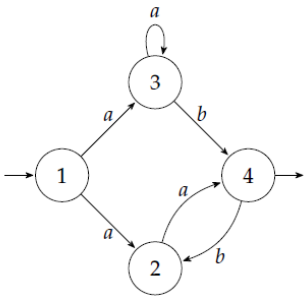
État	a	b
0	1	0
1	1	2
2	3	2
3	3	2

L'automate déterministe qui reconnaît les mots ayant le facteur ab

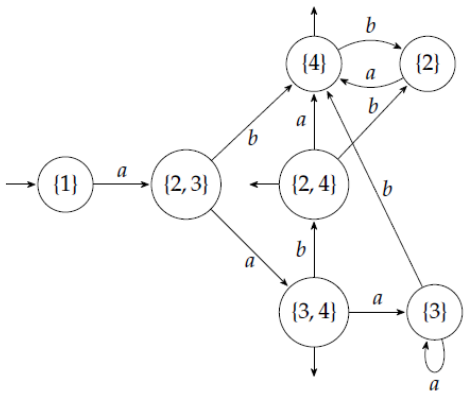


Note : Cet automate n'est pas évident à trouver mais grâce à l'algorithme de déterminisation, on peut le construire automatiquement.

Exemple 2 : Un automate à déterminiser



Le résultat de la déterminisation



Déterminisation avec les ε-transitions

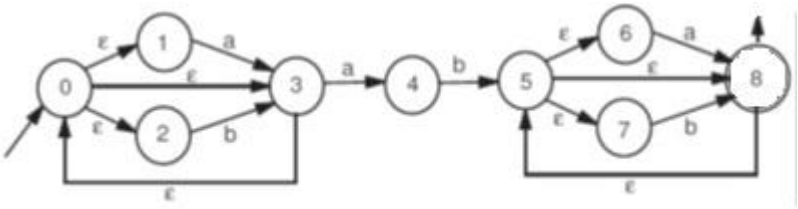
La déterminisation d'un AEF contenant au moins une ε-transition est un peu plus compliqué puisqu'elle fait appel à la notion de l'**ε-fermeture** d'un ensemble d'états.

Définition: Soit E un ensemble d'états. On appelle ε-fermeture de E l'ensemble des états incluant, en plus de ceux de E, tous les états accessibles depuis les états de E par un chemin étiqueté par le mot ε.

Exemple:

Considérons l'automate suivant, calculons un ensemble d'ε-fermetures :

L'automate reconnaissant les mots contenant le facteur ab (en ayant des ε-transitions)



– ε-fermeture ({0}) = {0, 1, 2, 3}

– ε-fermeture ({1, 2}) = {1, 2}

– ε-fermeture ({3}) = {0, 1, 2, 3}

Algorithme : Déterminisation d'un AEF comportant des "-transitions

Le principe de cet algorithme repose sur l'utilisation des ε-fermetures qui représenteront les états du nouvel automate.

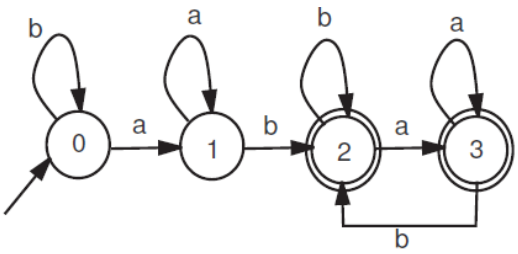
1- Partir de l'ε-fermeture de l'état initial (elle représente le nouvel état initial) ;
2- Rajouter dans la table de transition toutes les ε-fermetures des nouveaux états produits avec leurs transitions ;
3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel état ;
4- Tous les ε-fermetures contenant au moins un état final du premier automate deviennent finaux ;
5- Renuméroter les états en tant qu'états simples.

Appliquons maintenant ce dernier algorithme à l'automate précèdent

État	a	b
0,1,2,3	0,1,2,3,4	0,1,2,3
0,1,2,3,4	0,1,2,3,4	0,1,2,3,5,6,7,8
0,1,2,3,5,6,7,8	0,1,2,3,4,5,6,7,8	0,1,2,3,5,6,7,8
0,1,2,3,4,5,6,7,8	0,1,2,3,4,5,6,7,8	0,1,2,3,5,6,7,8

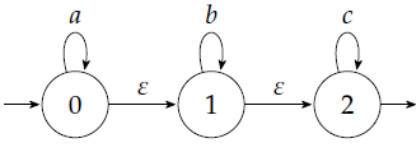
=>

État	a	b
0	1	0
1	1	2
2	3	2
3	3	2

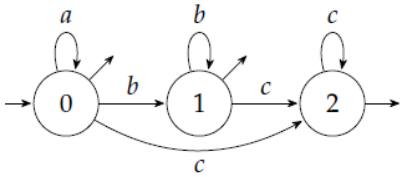


Exemple 2

Un automate avec ε-transitions correspondant à a*b*c* ($a^n b^m c^l$ avec n,m,l≥0)

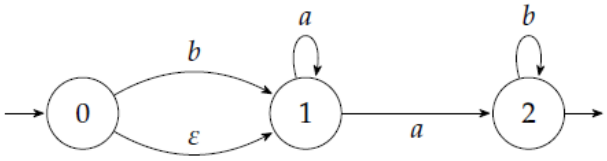


L'automate déterministe équivalent



Exercice

Construisez l'automate déterministe A' équivalent à A suivant.



2.10 Minimisation d'un AEF déterministe

D'une manière générale

- moins un automate contient d'états, moins il prendra du temps à reconnaître un mot (par exemple, en déterminisant un AEF, on tombe le plus souvent sur un automate comportant plus d'états qu'il n'en faut).
- De plus, il prendra moins d'espace en mémoire s'il est question de le sauvegarder.

→Il est donc logique de vouloir minimiser ce temps en essayant de réduire le nombre d'états. Si on peut trouver une multitude d'automates pour reconnaître le même langage, on ne peut trouver néanmoins qu'un seul automate minimal reconnaissant le même langage.

La minimisation s'effectue en éliminant les états dits ***inaccessibles*** et en *confondant* (ou fusionnant) les états reconnaissant le même langage.

Les états inaccessibles

Définition : Un état est dit inaccessible s'il n'existe aucun chemin permettant de l'atteindre à partir de l'état initial.

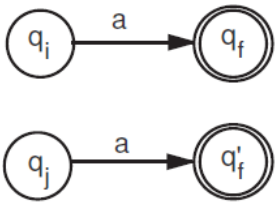
→Donc les états inaccessibles sont improductifs, c'est-à-dire qu'ils ne participeront jamais à la reconnaissance d'un mot.

Les états β-équivalents

Définition : Deux états q_i et q_j sont dits β-équivalents s'ils permettent d'atteindre les états finaux en utilisant les mêmes mots. On écrit alors : qi β qj.

Par le même mot, on entend que l'on lit la même séquence de symboles pour atteindre un état final à partir de qi et qj. Par conséquent, ces états reconnaissent le même langage.

La figure suivante montre un exemple d'états β-équivalents car l'état qi atteint les états finaux via le mot a, de même pour l'état qj. L'algorithme de minimisation consiste donc à fusionner simplement ces états pour n'en faire qu'un.



Remarque 1 : La relation β-équivalence est une relation d'équivalence. De plus, ∀x ∈ X (X étant l'alphabet), δ(qi, x) et δ(qj, x) sont également β-équivalents (puisque qi et qj reconnaissent le même langage). La relation β-équivalence est donc dite une relation de **congruence**.

Remarque 2 : Le nombre de classes d'équivalence de la relation β-équivalence est égal au nombre des états de l'automate minimal car les état de chaque classe d'équivalence reconnaissent le même langage (ils seront fusionnés).

Minimiser un AEF

La méthode de réduction d'un AEF est la suivante :

1. *Nettoyer* l'automate en éliminant les états inaccessibles ;
2. Regrouper les états congruents (appartenant à la même classe d'équivalence).

Algorithme : Regroupement des états congruents

Dans cet algorithme, chaque classe de congruence représentera un état dans l'automate minimal.

Il faut cependant noter que cet algorithme ne peut s'appliquer que si l'automate est déterministe.

Algorithme : Regroupement des états congruents

Dans cet algorithme, chaque classe de congruence représentera un état dans l'automate minimal. **Il faut cependant noter que cet algorithme ne peut s'appliquer que si l'automate est déterministe.**

1- Faire deux classes : A contenant les états finaux et B contenant les états non finaux ;
2- S'il existe un symbole a et deux états q_i et q_j d'une même classe tel que $\delta(q_i, a)$ et $\delta(q_j, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer q_i et q_j. On laisse dans la même classe tous les états qui donnent un état d'arrivée dans la même classe ;
3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de classes à séparer ;

Les paramètres de l'automate minimal sont, alors, les suivants :

- Chaque classe de congruence est un état de l'automate minimal ;
- La classe qui contient l'ancien état initial devient l'état initial de l'automate minimal ;
- Toute classe contenant un état final devient un état final ;

- La fonction de transition est définie comme suit : soient A une classe de congruence obtenue, a un symbole de l'alphabet et qi un état qi ∈ A tel que δ(qi, a) est définie. La transition δ(A, a) est égale à la classe B qui contient l'état qj tel que δ(qi, a) = qj.

Exemple 13 : Soit à minimiser l'automate suivant (les états finaux sont les états 1 et 2 tandis que l'état 1 est initial) :

État	a	b
1	2	5
2	2	4
3	3	2
4	5	3
5	4	6
6	6	1
7	5	7

1. La première étape consiste à éliminer les états inaccessibles, il s'agit juste de l'état 7.

Les étapes de détermination des classes de congruences sont les suivantes :
A = {1, 2}, B = {3, 4, 5, 6} ;

2. δ(3, *b*) = 2 ∈ A,δ(4, *b*) = 3 ∈ B ainsi il faut séparer 4 du reste de la classe B. Alors, oncrée une classe C contenant l'état 4 ;
3. δ(3, *b*) = 2 ∈ A, δ(5, *b*) = 6 ∈ A ainsi il faut séparer 5 du reste de la classe B. Mais inutile de créer une autre classe puisque
   δ(4, *a*) = 5 ∈ B,
   δ(5, *a*) = 4 ∈ B
   et δ(4, *b*) = 3 ∈ B
   et δ(5, *b*) = 6 ∈ B,
   il faut donc mettre 5 dans la classe C. C = {4, 5} et B = {3, 6} ;

4. Aucun autre changement n'est possible, alors on arrête l'algorithme.

Le nouvel automate est donc le suivant (l'état initial est A) :

Etat	a	b
A	A	C
B	B	A
C	C	A

L'automate obtenu est **minimal** et est **unique**,

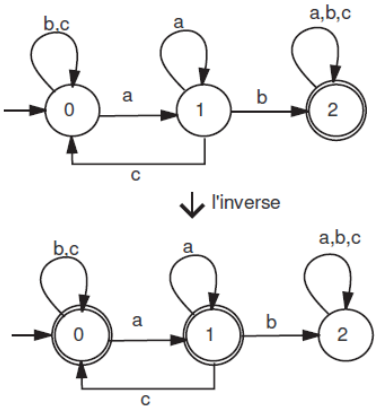
2.11 Opérations sur les automates

2.11.1 Le complément

Soit A= (A,Q, q_0, Q_F, δ) un automate déterministe reconnaissant le langage ***L***. L'automate reconnaissant le langage inverse (c'est-à-dire L ou A* −*L*) est défini par le quintuplet (A,Q, q_0,Q − Q_F, δ) (en d'autres termes, les états finaux deviennent non finaux et vice-versa).

Cependant, cette propriété ne fonctionne que si l'automate est complet : la fonction δ est complètement définie pour toute paire (q, a) ∈ Q×A comme le montre les exemples suivants.

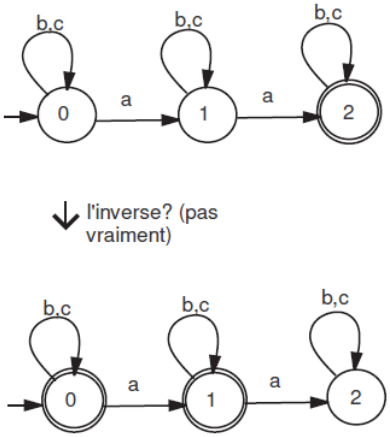
Exemple :
Soit le langage des mots définis sur l'alphabet {a, b, c} **<u>contenant le facteur ab</u>**. Il s'agit ici d'un automate complet, on peut, donc, lui appliquer la propriété précédente pour trouver l'automate des mots qui **<u>ne contiennent</u> pas le facteur ab**



Remarque : Si l'automate n'est pas complet, on ne peut pas obtenir l'automate du langage inverse.

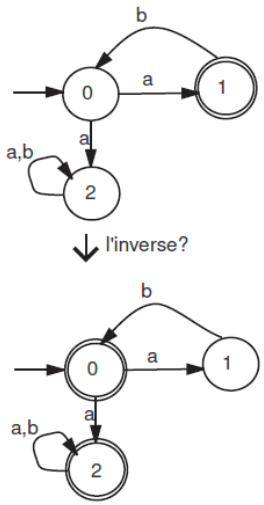
Exemple

Soit le langage des mots définis sur {a, b, c} contenant exactement deux a
L'automate n'est pas complet, donc, on ne peut pas appliquer la propriété précédente à moins de le transformer en un automate complet. Pour le faire, il suffit de rajouter un état non final E (on le désignera comme un état d'erreur) tel que δ(2, a) = E ; δ(E, a) = E ; δ(E, b) = E ; δ(E, c) = E ;



L'automate obtenu reconnaît les mots contenant au plus 1 a

Remarque : Si l'automate n'est pas déterministe, on ne peut pas trouver l'automate du langage complémentaire.
Exemple :



2.11.2 Produit d'automates

Définition : Soit X = (A,Q, q_0, Q_F, δ) et X′ = (A′,Q′, q_0', Q'_F, δ′) deux automates à états finis.
On appelle produit des deux automates X et X′ l'automate X′′(A′′,Q′′, q_0'' ,Q_F'', δ′′) défini comme suit :

– A′′ = A ∪ A′ ;
– Q′′ = Q × Q′ ;
– $q_0'' = (q_0, q_0')$;
– $Q_F'' = Q_F \times Q_F'$;

– $\delta((q, q'), a) = (\delta(q, a), \delta(q', a))$

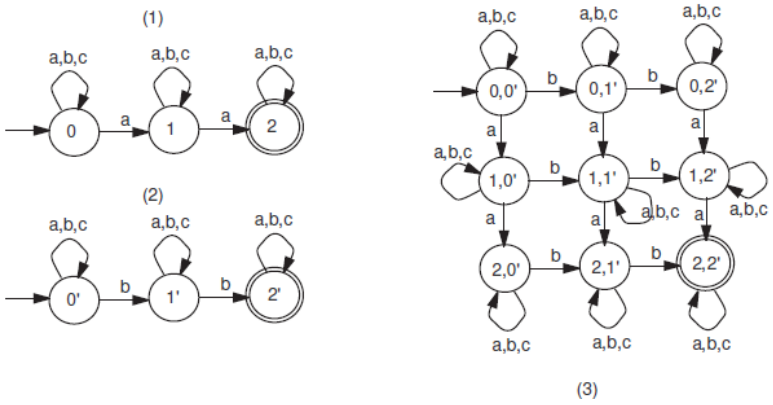
Cette définition permet de synchroniser la reconnaissance d'un mot par les deux automates.

On pourra facilement démontrer que si w est un mot reconnu par X'' alors il l'est par X, ceci est également le cas pour l'automate X'. Ainsi, si L(X) est le langage reconnu par X et L(X') est le langage reconnu par le langage X' alors l'automate X'' reconnaît l'intersection des deux langages : $L(X) \cap L(X')$.

Exemple

- L'automate (1) reconnaît les mots sur {a, b, c} contenant deux a ;
- L'automate (2) reconnaît les mots sur {a, b, c} contenant deux b,
- L'automate (3) représente le produit des deux automates.

Remarquons que dans l'automate (3), tout chemin qui mène de l'état initial vers l'état final passe forcément par deux a et deux b (tout ordre est possible). Or, ceci est exactement le langage résultant de l'intersection des deux premiers langages.



2.11.3 Le langage miroir

Soit $X = (A, Q, q_0, Q_F, \delta)$ un automate reconnaissant le langage L(X). L'automate qui reconnaît le langage $(L(X))^R$ est reconnu par l'automate $X^R = (A, Q, Q_F, \{q_0\}, \delta^R)$ tel que : $\delta^R(q', a) = q$ si $\delta(q, a) = q'$.

En d'autres termes, il suffit juste d'inverser les sens des arcs de l'automate et le statut initial/final des états initiaux et finaux. Notons, par ailleurs, que l'automate X^R peut contenir plusieurs états initiaux ce qui signifie qu'il est le plus souvent non déterministe. Pour éliminer le problème des multiples états initiaux, il suffit de rajouter un nouvel état initial et de le raccorder aux anciens états finaux par des ε-transitions.

Partie 3 : Langages réguliers

Type d'un langage

Un langage pouvant être engendré par une grammaire de type x et pas par une grammaire d'un type supérieur dans la hiérarchie, est appelé un langage de type ***x***.

Type	Nom
3	régulier
2	hors contexte (algébrique)
1	dépendant du contexte
0	récursivement énumérable

Les langages réguliers

Les langages réguliers sont les langages générés par des grammaires de type 3 (ou encore grammaires régulières). Ils sont reconnus grâce aux automates à états finis. Le terme régulier vient du fait que les mots de tels langages possèdent une forme particulière pouvant être décrite par des *expressions* dites ***régulières***.

Ce chapitre introduit ces dernières et établit l'équivalence entre les trois représentations : ***expression régulière***, ***grammaire régulière*** et ***AEF***.

Exemples

- $L_1 = \{m \in \{a, b\}^*\}$
  ➔ $G_1 = (\{S\}, \{a, b\}, \{S \rightarrow aS \mid bS \mid \varepsilon\}, S)$

- $L_2 = \{m \in \{a, b\}^* / |m|_a \bmod 2 = 0\}$
  ➔ $G_2 = (\{S, T\}, \{a, b\}, \{S \rightarrow aT \mid bS \mid \varepsilon, T \rightarrow aS \mid bT\}, S)$

- $L_3 = \{m \in \{a, b\}^* / m = xaaa \text{ avec } x \in \{a, b\}^*\}$
  ➔ $G_3 = (\{S, T,U\}, \{a, b\},\{S \rightarrow aS \mid bS \mid aT, T \rightarrow aU, U \rightarrow a\}, S)$

- $L_4 = \{m \in \{a, b\}^* \mid |m|_a \bmod 2 = 0 \text{ et } |m|_b \bmod 2 = 0\}$
  ➔ $G_4 = (\{S, T,U,V\}, \{a, b\},\{ S \rightarrow aT \mid bU \mid \varepsilon, T \rightarrow aS \mid bV, V \rightarrow aU \mid bT, U \rightarrow aV \mid bS\}, S)$

Expressions régulières et langages associes

La notion d'expression régulière est d'usage fréquent en informatique. En effet, on a souvent recours aux expressions régulières lorsqu'on désire rechercher certains motifs récurrents

Expressions régulières

Les expressions régulières permettent de décrire les langages réguliers, de façon plus simple qu'en utilisant des opérations ensemblistes.

Définition (*expression régulière*) : Une expression E est une expression régulière sur un alphabet A si et seulement si

- E = ϕ ou
- E = ε ou
- E = a avec $a \in$A ou
- E = E_1 | E_2 (ou E= E_1+E_2)et E_1 et E_2 sont deux expressions régulières sur A
- E = E_1.E_2 et E_1 et E_2 sont deux expressions régulières sur A ou
- E = E_1* et E_1 est une expression régulière sur A

Les opérateurs *****, **.** et **|** ont une priorité décroissante. Si nécessaire, on peut ajouter des parenthèses.

Définition (langage décrit par une expression régulière) : le langage $L(E)$ décrit par une expression régulière E définie sur un alphabet A est défini par

- $L(E)$ = ϕ si E = ϕ,
- $L(E)$ = {ε} si E = ε,
- $L(E)$ = {a} si E = a,
- $L(E) = L(E_1) \cup L(E_2)$ si $E = E_1 | E_2$,
- $L(E) = L(E_1).L(E_2)$ si $E = E_1.E_2$,
- $L(E) = L(E_1)^*$ si $E = E_1^*$ où E_1 est une expression régulière sur A.

Par exemple, sur l'alphabet A = {a, b, c},
- E_1 = a*bbc* décrit le langage L(E_1) = {$a^n bbc^p$ /n ≥ 0, p ≥ 0}
- E_2 = (a | b | c)*(bb |cc)a* décrit le langage L(E_2) = {$wbba^n$, $wcca^n$ / w ∈ A*; n ≥ 0}

Remarque : L'équivalence entre expressions régulières et langages réguliers est établie par les deux théorèmes suivants.

Théorème : Toute expression régulière décrit un langage régulier.

Théorème : Tout langage régulier peut être décrit par une expression régulière

Quelques propriétés des langages réguliers

Théorème : Soient $\boldsymbol{L}_1$ et $\boldsymbol{L}_2$ deux langages réguliers. Les langages $\boldsymbol{L}_1 \cup \boldsymbol{L}_2$, $\boldsymbol{L}_1 \cap \boldsymbol{L}_2$, $\boldsymbol{L}_1.\boldsymbol{L}_2$,$C(\boldsymbol{L}_1)$ et $\boldsymbol{L}_1^*$ sont des langages réguliers.

Les langages réguliers, les grammaires et les automates à états finis

Le théorème suivant établit l'équivalence entre les AEF, les grammaires régulières et les expressions régulières :

Théorème : (Théorème de Kleene) Soient Λ_{reg} l'ensemble des langages réguliers (générés par des grammaires régulières), Λ_{rat} l'ensemble des langages décrits par toutes les expressions régulières et Λ_{AEF} l'ensemble de tous les langages reconnus par un AEF. Nous avons, alors, l'égalité suivante :

$$\Lambda_{reg} = \Lambda_{rat} = \Lambda_{AEF}$$

Le théorème annonce que l'on peut passer d'une représentation à une autre du fait de l'équivalence entre les trois représentations. Les sections suivantes expliquent comment passer d'une représentation à une autre.

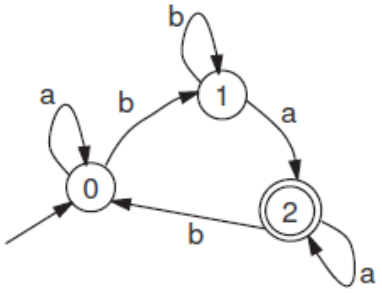
Passage de l'automate vers l'expression régulière

Soit X = (A,Q, q_0, Q_F, δ) un automate à états fini quelconque. On note par L_i le langage reconnu par l'automate si son état initial était q_i. Par conséquent, trouver le langage reconnu par l'automate revient à trouver L_0 étant donné que la reconnaissance commence à partir de l'état initial q_0. L'automate permet d'établir un système d'équations aux langages de la manière suivante :

– si $\delta(q_i, a) = q_j$ alors on écrit : $L_i = aL_j$;
– si $q_i \in Q_F$, alors on écrit : $L_i = \varepsilon$
– si $L_i = \alpha$ et $L_i = \beta$ alors on écrit : $L_i = \alpha|\beta$;

Il suffit ensuite de résoudre le système en précédant à des substitutions et en utilisant la règle suivante : la solution de l'équation $L = \alpha L|\beta$ ($\varepsilon \notin \alpha$) est le langage $L = \alpha^*\beta$

Exemple : Soit l'automate donné par la figure suivante :



Trouvons le langage reconnu par cet automate. Le système d'équations est le suivant :

– 1) $L_0 = aL_0|bL_1$;
– 2) $L_1 = bL_1|aL_2$;
– 3) $L_2 = aL_2|bL_0|\varepsilon$

Appliquons la deuxième règle sur les équations, on obtient alors après substitutions :

– 4) $L_0 = a^*bL_1$;
– 5) $L_1 = b^*aL_2 = b^*a+bL_0|b^*a^+$;
– 6) $L_2 = a^*bL_0|a^*$

En remplaçant 5) dans 4) on obtient : $L_0 = a^*b^+a^+bL_0|a^*b^+a^+$ ce qui donne alors $L_0 = (a^*b^+a+b)^*a^*b^+a^+$ (remarquez que l'on pouvait déduire la même chose à partir de l'automate...).

Passage de l'expression régulière vers l'automate

Il existe deux méthodes permettant de réaliser cette tâche. La première fait appel à la notion de dérivée tandis que la deuxième construit un automate comportant des ε-transitions en se basant sur les propriétés des langages réguliers.

La méthode des dérivées

Définition:
Soit *w* un mot défini sur un alphabet A. On appelle dérivée de *w* par rapport à $u \in A^*$ le mot $v \in A^*$ tel que $w = uv$. On note cette opération par : $w||u = v$.

On peut étendre cette notion aux langages. Ainsi la dérivée d'un langage par rapport à un mot $u \in A^*$ est le langage $L||u = \{v \in A^* | \exists w \in L : w = uv\}$.

Exemple :
– $L = \{1, 01, 11\}$, $L||1 = \{\varepsilon, 1\}$, $L||0 = \{1\}$, $L||00 = \emptyset$

Propriétés des dérivées

– $(\sum_{i=1}^{n} L_i)||u = \sum_{i=1}^{n}(L_i||u)$;
– $L.L'||u = (L||u).L' + f(L).(L'||u)$ tel que $f(L) = \{\varepsilon\}$ si $\varepsilon \in L$ et $f(L) = \emptyset$ sinon ;
– $(L^*)||u = (L||u)L^*$.

Méthode de construction de l'automate par la méthode des dérivées

Soit ***r*** une expression régulière (utilisant l'alphabet A) pour laquelle on veut construire un AEF. L'algorithme suivant donne la construction de l'automate :

1. Dériver ***r*** par rapport à chaque symbole de A ;
2. Recommencer 1) pour chaque nouveau langage obtenu jusqu'à ce qu'il n'y ait plus de nouveaux langages ;
3. Chaque langage obtenu correspond à un état de l'automate. L'état initial correspond à *r*. Si ε appartient à un langage obtenu alors l'état correspondant est final ;
4. si $L_i||a = L_j$ alors on crée une transition entre l'état associé à L_i et l'état associé à L_j et on la décore par a.

Exemple : Considérons le langage (a|b)*a(a|b)*. On sait que :
– (a|b)||a = ε donc (a|b)*||a = (a|b)*

Commençons l'application de l'algorithme :
– [(a|b)*a(a|b)*]||a = ((a|b)*a(a|b)*)|(a|b)* ;
– [(a|b)*a(a|b)*]||b = (a|b)*a(a|b)* ;
– [((a|b)*a(a|b)*)|(a|b)*]||a = ((a|b)*a(a|b)*)|(a|b)* ;
– [((a|b)*a(a|b)*)|(a|b)*]||b = ((a|b)*a(a|b)*)|(a|b)* ;

Il n'y a plus de nouveaux langages, on s'arrête alors. L'automate comporte deux états q_0 (associé à (a|b)*a(a|b)*) et q_1 (associé à ((a|b)*a(a|b)*)|(a|b)*), il est donné par la table suivante (l'état initial est q_0 et l'état q_1 est final) :

État	a	b
q_0	q_1	q_0
q_1	q_1	q_1

La méthode des dérivée ne permet pas seulement de construire l'AEF d'un langage, elle permet même de vérifier si un langage est régulier ou non. Pour cela, nous allons accepter le théorème suivant :

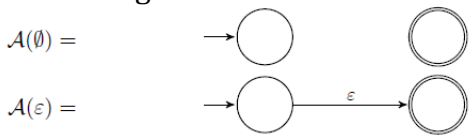
Théorème :
Un langage est régulier si et seulement si le nombre de langages trouvés par la méthode des dérivées est fini. En d'autres termes, l'application de la méthode des dérivées à un langage non régulier produit un nombre infini des langages (ou encore d'états).

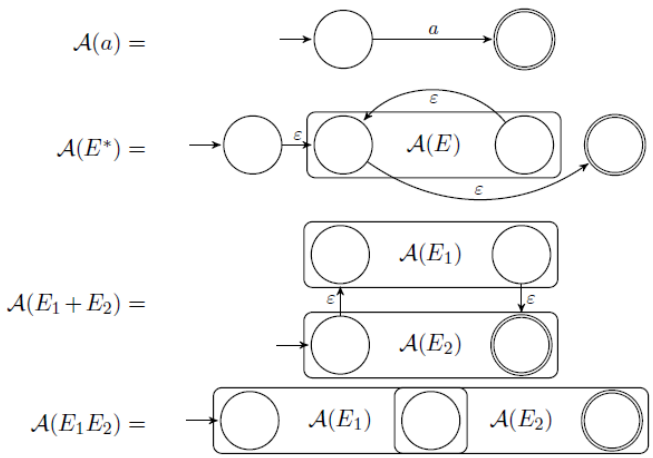
1. AFN associé à une expression régulière (algorithme de Thompson)

On construit par induction un AFN avec ε–transition reconnaissant le langage décrit par une expression rationnelle, à partir des opérations de combinaison définissant les expressions rationnelles (union, concaténation, étoile).

- Pour ∅, ε, et les lettres, l'automate est évident.
- Pour l'union, la concaténation et l'étoile, on combine les AFN comme cela est indiqué dans la preuve du théorème de Kleene.
- Les automates générés ont un unique état initial, et un unique état final

Soient E_1 et E_2 deux expressions régulières





La taille d'une expression rationnelle r, notée |r|, est son nombre de caractères (sans compter les parenthèses).

Exemple : |(a + ab)*(ε + ab)| = 9.

NB. L'algorithme de Thompson construit à partir d'une expression régulière r un AFN avec ε–transitions où :

- il y a un état initial et un état final,
- le nombre d'états est borné par 2|r|,
- de chaque état sort au plus deux transitions

2. AFN associé à une expression régulière (algorithme de Glushkov)

- ***Première étape*** : Linéariser l'expression rationnelle, en remplaçant toutes les lettres par des symboles distincts (de la gauche vers la droite).
  Exemple : pour l'expression régulière (a + ab)*(ε + ab)
  $(x_1 + x_2x_3)^*(\varepsilon + x_4x_5)$

- ***Deuxième étape*** : Déterminer
  - ***Premier*** = ensemble des symboles pouvant commencer un mot = $\{x_1, x_2, x_3\}$,
  - ***Dernier*** = ensemble des symboles pouvant finir un mot = $\{x_4, x_5\}$,
  - Pour tout symbole x_i, ***Suivant***(x_i) = ensemble des symboles pouvant suivre x_i.

Attention : cette fonction ***Suivant*** doit tenir compte des étoiles.

	Suivant
x_1	x_1, x_2, x_4
x_2	x_3
x_3	x_1, x_2, x_4
x_4	x_5
x_5	

Proposition. Un mot $w = w_1w_2 \ldots w_n$ appartient au langage décrit par l'expression linéarisée ssi $w_1 \in$ *Premier*, $w_n \in$ *Dernier*, et $w_{i+1} \in$ *Suivant*(w_i) pour i =1, … n-1.

- ***Construction de l'AFN*** pour l'expression linéarisée **:**
  - **Q** = ensemble des symboles + un état initial ***i***
  - $\mathbf{Q_F}$ = *Dernier*, avec en plus l'état initial ***i*** si **ε** appartient au langage
  - fonction de ***transition* δ** :

    $\delta(i, x_i) = x_i$ si $x_i \in$ Premier,

    $\delta(x_i, x_j) = x_j$, pour tout $x_j \in$ *Suivant*(x_i).

- Puis on remplace les symboles x_i par les lettres d'origine de l'expression rationnelle.

Exemple: (ab + c)∗ab est linéarisée en (12 + 3)∗45. On a α1 = α4 = a, α2 = α5 = b et α3 = c.

• On définit un état pour chaque position i de l'expression, et on ajoute un état initial 0.

• On place une transition de i vers j étiquetée par αj si et seulement si la "lettre" j peut suivre la lettre i dans un mot du langage L(E′).

• On place une transition de 0 vers i si et seulement si la "lettre" i peut être la première lettre d'un mot du langage L(E′).

• L'état i (i > 0) est final si et seulement si i peut être la dernière "lettre" d'un mot du langage L(E′).

• L'état 0 est final si et seulement si le mot vide appartient à L(E′).

L'automate ainsi construit reconnaît le langage L(e) associé à l'expression initiale. L'automate de Glushkov de l'expression donnée en exemple possède 6 états. L'état 0 est initial, l'état 5 est seul état final. Sa table de transition est la suivante :

	a	b	c
0	{1, 4}	∅	{3}
1	∅	{2}	∅
2	{1, 4}	∅	{3}
3	{1, 4}	∅	{3}
4	∅	{5}	∅
5	∅	∅	∅

Exemple :

Construire les automates non-déterministes sans ε-transitions qui acceptent les $L(E_i)(i=1..5)$, en appliquant la construction de Gluskov

E_1=(a + ab)*(ε + ab)
E_2=(a*b*)*ab
E_3=a(b+ab)*+b*(a+bb)
E_4=b(ab)*+(ba)*b
E_5=(ab+c)*ab

Passage de la grammaire vers l'automate

D'après la section précédente, il existe une forme de grammaires régulières pour lesquelles il est facile de construire l'AEF correspondant. En effet, soit G = (V,N, S, R) une grammaire régulière à droite, si toutes les règles de production sont de la forme : $A \rightarrow aB$ ou $A \rightarrow B$
$(A, B \in N, a \in V \cup \{\varepsilon\})$ alors il suffit d'appliquer l'algorithme suivant :
1. Associer un état à chaque non terminal de N;
2. L'état initial est associé à l'axiome ;
3. Pour chaque règle de production de la forme $A \rightarrow \varepsilon$, l'état q_A est final ;
4. Pour chaque règle de production de la forme $A \rightarrow a$ $(a \in V)$, alors créer un nouvel état final q_f et une transition partant de l'état q_A vers l'état q_f avec l'entrée a ;
5. Pour chaque règle $A \rightarrow aB$ alors créer une transition partant de q_A vers l'état q_B en utilisant l'entrée a ;
6. Pour chaque règle $A \rightarrow B$ alors créer une ε-transition partant de q_A vers l'état q_B ;

Quelques propriétés des langages réguliers

Théorème : Soient L_1 et L_2 deux langages réguliers. Les langages $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1.L_2$, $c(L_1)$ et L_1^* sont des langages réguliers.

4 Partie 04 : Langages hors-contexte (algébriques) et Automates à pile

Certains langages ne peuvent pas être décrits par une grammaire régulière, et ne peuvent donc pas être reconnus par un automate fini (par exemple le langage $\{a^n b^n / n > 0\}$). On étudie dans ce chapitre une classe de langages plus générale que celle des langages réguliers : la classe des langages hors-contexte, décrits par des grammaires hors-contexte et reconnus par des automates à pile.

Définition (***Grammaire hors-contexte***) : G = (T, N, S, *R*) est une grammaire hors-contexte si toutes les règles de *R* sont de la forme $A \rightarrow w$ avec $A \in N$ et $w \in (N \cup T)^*$.

Définition (***Langage hors-contexte***) : On appelle langage hors-contexte un langage généré par une grammaire hors contexte.

Automates à pile

Les langages hors-contexte, décrits par des grammaires hors-contexte, sont reconnus (acceptés) par des automates à pile. De façon informelle, un automate à pile est un automate fini auquel on a ajouté une pile de capacité illimitée initialement vide. L'exécution d'un automate à pile sur un mot donné est semblable à celle d'un automate fini. Toutefois, à chaque étape, l'automate à pile consulte le sommet de sa pile et le remplace éventuellement par une suite de symboles.

Définition (***Automate à pile***) : Un automate à pile est un quintuplet $A = (T,P,Q,\delta,q_0)$ tel que :

– T est le vocabulaire terminal,
– P est le vocabulaire de pile (contenant en particulier un symbole initial de pile vide, noté Z),
– Q est un ensemble fini d'états,
– δ est un ensemble de transitions,
– $q_0 \in Q$ est l'état initial.

……..

4.4 Simplification des grammaires hors-contextes

4.4.1 Les grammaires propres

Une grammaire hors-contexte (V, N, S, R) est dite propre si elle vérifie :

- $\forall A \rightarrow u \in R : u \neq \varepsilon$ ou $A = S$;
- $\forall A \rightarrow u \in R$: S ne figure pas dans u ;
- $\forall A \rightarrow u \in R : u \notin N$;
- Tous les non terminaux sont utiles, c'est-à-dire qu'ils vérifient :
  - $\forall A \in N$: A est atteignable depuis S : $\exists \alpha,\beta \in (N + V)^* : S \rightarrow^* \alpha A \beta$;
  - $\forall A \in N$: A est productif : $\exists w \in V^* : A \rightarrow^* w$.

Il est toujours possible de trouver une grammaire propre pour toute grammaire hors contexte.

En effet, on procède comme suit :

1. Rajouter une nouvelle règle $S' \rightarrow S$ tel que S' est le nouvel axiome ;
2. **Éliminer** les règles $\mathbf{A \rightarrow \varepsilon}$:
   - Calculer l'ensemble $E = \{A \in N \cup \{S'\} | A \rightarrow^* \varepsilon\}$;
   - Pour tout $A \in E$, pour toute règle $B \rightarrow \alpha A \beta$ de R
   - Rajouter la règle $B \rightarrow \alpha\beta$
   - Enlever les règles $A \rightarrow \varepsilon$;
3. **Eliminer** les règles $\mathbf{A \rightarrow^* B}$, on applique la procédure suivante sur R privée de $S' \rightarrow \varepsilon$:
   - Calculer toutes les paires (A, B) tel que $A \rightarrow^* B$
   - Pour chaque paire (A, B) trouvée
     - Pour chaque règle $B \rightarrow u_1|...|u_n$ rajouter la règle $A \rightarrow u_1|...|u_n$
   - Enlever toutes les règles $A \rightarrow B$
4. **Supprimer** tous les non-terminaux **non-productifs**
5. **Supprimer** tous les non-terminaux **non-atteignables**

4.4.2 Les formes normales

La forme normale de Chomsky

Soit G = (V,N, S, R) une grammaire hors-contexte. On dit que G est sous forme normale de Chomsky si les règles de G sont toutes de l'une des formes suivantes :

- $A \rightarrow BC$, $A \in N$, $B,C \in N - \{S\}$
- $A \rightarrow a$, $A \in N$, $a \in V$
- $S \rightarrow \varepsilon$

L'intérêt de la forme normale de Chomsky est que les arbres de dérivations sont des arbres binaires ce qui facilite l'application de pas mal d'algorithmes.

Il est toujours possible de transformer n'importe quelle grammaire hors-contexte pour qu'elle soit sous la forme normale de Chomsky. Notons d'abord que si la grammaire est propre, alors cela facilitera énormément la procédure de transformation. Par conséquent, on suppose ici que la grammaire a été rendue propre. Donc toutes les règles de S sont sous l'une des formes suivantes :

- $S \rightarrow \varepsilon$
- $A \rightarrow w$, $w \in V+$
- $A \rightarrow w$, $w \in ((N - \{S\}) + V)^*$

La deuxième forme peut être facilement transformée en $A \rightarrow BC$. En effet, si $w = au$, $u \in V^+$

alors il suffit de remplacer la règle par les trois règles $A \rightarrow A_1A_2$, $A_1 \rightarrow a$ et $A_2 \rightarrow u$. Ensuite,
il faudra transformer la dernière règle de manière récursive tant que $|u| > 1$.
Il reste alors à transformer la troisième forme. Supposons que :

$$w = w_1A_1w_2A_2...w_nA_nw_{n+1} \text{ avec } w_i \in V^* \text{ et } A_i \in (N - \{S\})$$

La procédure de transformation est très simple.
Si $w \neq \varepsilon$ alors il suffit de transformer cette règle en :

$$A \rightarrow B_1B_2$$

$B_1 \rightarrow w_1$

$B_2 \rightarrow A_1w_2A_2...w_nA_nw_{n+1}$

sinon, elle sera transformée en :

$A \rightarrow A_1B$

$B \rightarrow w_2A_2...w_nA_nw_{n+1}$

Cette transformation est appliquée de manière récursive jusqu'à ce que toutes les règles soient des règles normalisées.

Exemple : Soit la grammaire dont les règles de production sont : S → aSbS | bSaS | ε, on veut obtenir sa forme normalisée de Chomsky.

La forme normale de Greibach

Soit G = (V,N, S, R) une grammaire hors-contexte. On dit que G est sous la forme normale de Greibach si toutes ses règles sont de l'une des formes suivantes :

- $A \rightarrow aA_1A_2...A_n$, $a \in V$, $A_i \in N - \{S\}$
- $A \rightarrow a$, $a \in V$
- $S \rightarrow \varepsilon$

L'intérêt pratique de la mise sous forme normale de Greibach est qu'à chaque dérivation, on détermine un préfixe de plus en plus long formé uniquement de symboles terminaux. Cela permet de construire plus aisément des analyseurs permettant de retrouver l'arbre d'analyse associé à un mot généré. Cependant, la transformation d'une grammaire hors-contexte en une grammaire sous la forme normale de Greibach nécessite plus de travail et de raffinement de la grammaire. Nous choisissons de ne pas l'aborder dans ce cours.

Exemple 1:

Mettre sous forme normale de Chomsky la grammaire définie par les règles de production suivantes : S →AB | aS | a

A → Ab | ε

B → AS

Exemple 2:

Considérer la grammaire G suivante :

Axiome = S

N = {S; A;B;C} T = {a; b} P = {S → aAa ; A → Sb|bBB ; B → abb|aC ; C → aCA}

1. Mettre la grammaire G sous forme normale de Chomsky.
2. Transformer cette même grammaire G afin qu'elle soit sous forme normale de Greibach.