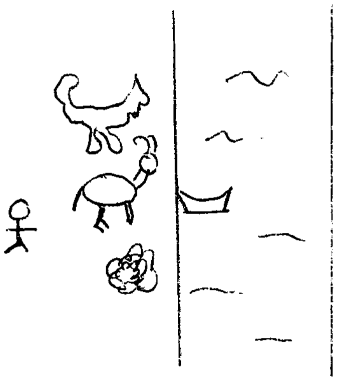
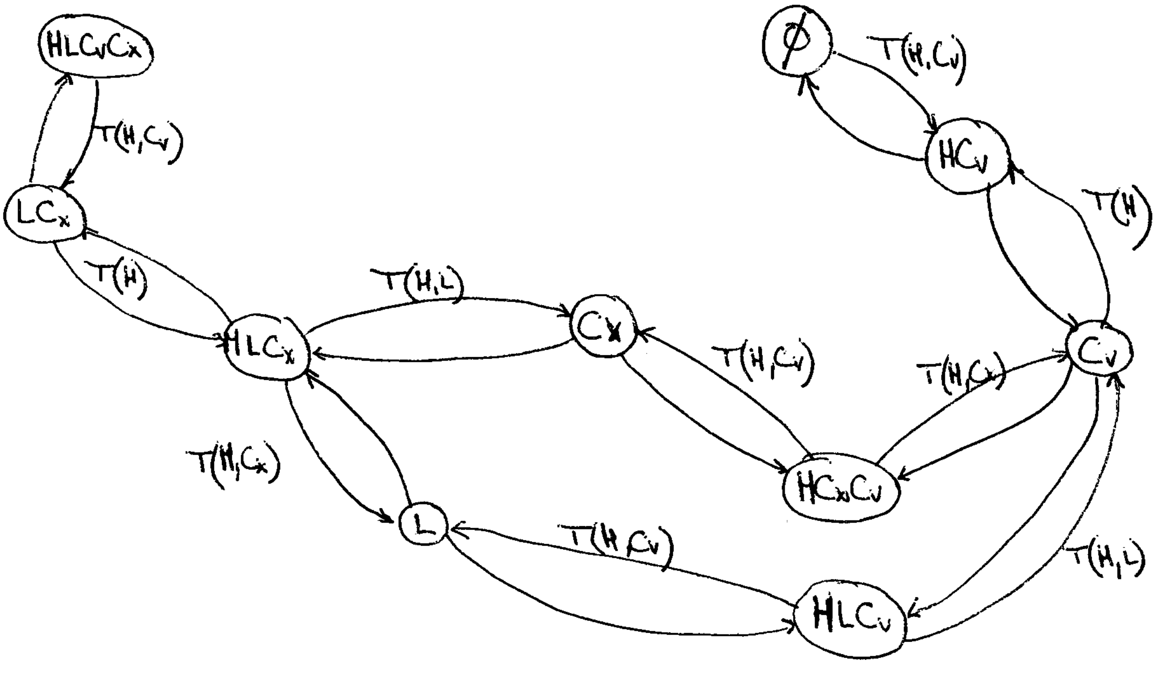
Révisions

…n va faire quelques rappels sur les automates finis.

…emple : le Loup, Chou, Chèvre.



…s états de l'automate sont les configurations de la situation. On passera d'un état à un autre grâce …x actions qui seront la traversée par au plus 2 personnages.
…n aura donc l'automate suivant :



Rmq : Ce graphe n'est pas complet. Les états listent les personnages sur la rive gauche.

…Q : états – ensembles finis. (états)
…Σ : alphabet – ensembles finis (action).
…$q_0 \in Q$: état initial
…$F \subseteq Q$. ensemble d'états finaux.
…$(\Delta) \subset (Q \times \Sigma \times Q)$: fonction de transition (ou relation de transition).
…n automate est donc un 5-uplets $A = (Q, \Sigma, q_0, F, \delta)$.
…n calcul (Run, exécution) d'un automate est une séquence : $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} p_n$ où les p_i sont …s états, les a_i sont des actions. L'état de départ est p_i et $\forall i$, $(p_i, a_{i+1}, p_{i+1}) \in \delta$. On peut aussi

ire que c'est un calcul sur le mot $a_1, a_2, \dots, a_n$.
Un calcul est accepteur si $p_n \in F$.
Un mot est accepté (reconnu) par l'automate ssi il y a un calcul accepteur sur ce mot.
Le langage $L(A)$ reconnu par l'automate A est l'ensemble de tous les mots acceptés.
Dans le cas d'un automate comportemental, on parle, dans le cas d'un calcul, d'un comportement possible. Un calcul accepteur est donc un comportement qui mène à la fin. Un mot est accepté s'il résout le problème. Enfin on ne parle pas de langage mais d'ensembles de solutions possibles.

On a donc défini les automates finis non-déterministes. Pour qu'un automate soit déterministe, il faut que la relation Δ soit une fonction $\forall q \in Q, \forall a \in \Sigma, \exists! q' \in Q$ tq $(q, a, q') \in \Delta$. (on a également mis la définition d'un automate complet ici).

Théorème : Pour chaque automate non-déterministe, il existe un automate déterministe qui accepte le même langage.
Rq : Si A un automate non-déterministe à n états, alors son équivalent A', déterministe a, au plus, 2^n états.
Preuve : Construction de sous-ensembles. Etats de A' = ensemble d'états possibles de A.

Une extension peut être un automate avec une ε-transition. C'est-à-dire que dans la relation de transition, on peut avoir des transitions étiquetées par $a \in \Sigma \cup \{\varepsilon\}$. Une ε-transition est une transition muette : elle ne consomme pas de lettres. On a alors le théorème précédent qui peut s'appliquer.

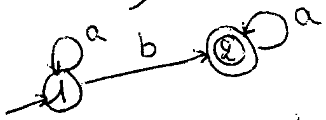
Propriétés des langages reconnaissables : Si L et M sont reconnus, alors $L \cup M$, $L \cap M$, $\overline{L}$, $L \setminus M$, $L.M$, L^*, miroir(L) sont également reconnaissables.
Les langages de base tels que $\emptyset$, $\{\varepsilon\}$, $\{a\}$ ... sont reconnus.
Méthodes de preuves : Il y a plusieurs solutions. On peut soit construire les automates pour les langages de base. Pour les combinaisons, on suppose que l'on a un automate pour L, un automate pour M et on les utilise pour construire un automate reconnaissant la combinaison voulue.
Remarque : - Pour obtenir un automate qui reconnait $\overline{L}$, il ne faut pas oublier de déterminiser et complémenter A avant d'échanger les états finaux en non-finaux et vice-versa.
- la solution triviale pour $L \cap M = \overline{\overline{L} \cup \overline{M}}$ est inefficace. Il vaut mieux utiliser $L(A \times B)$
- Pour obtenir $L \cup M$, on a 2 solutions : la première, naïve est de faire un automate non-déterministe qui fait ou L ou bien M. La seconde utilise $L(A \times B)$ en faisant fonctionner les 2 automates en // et de regarder si au moins un des deux répond 'oui'.

Une expression régulière (resp. rationnelle) sur Σ est E définie par : $E = \emptyset \mid \varepsilon \mid a \mid (E+E) \mid (E.E) \mid E^*$.
Chaque expression définit un langage tel que : $[\emptyset] = \emptyset$; $[\varepsilon] = \{\varepsilon\}$; $[a] = \{a\}$; $[f+g] = [f] \cup [g]$; $[f.g] = \{uv \mid u \in [f], v \in [g]\}$; $[f^*] = \{u_1 \dots u_n \mid \forall i, u_i \in [f]\}$.
L est un langage rationnel s'il existe une expression rationnelle qui le définit.
Théorème de Kleene. 'L est rationnel' est équivalent à 'L est reconnaissable'.
Preuve : . L rationnel $\Rightarrow$ L reconnaissable a déjà été vu. On connait les automates pour les 3 expressions rationnelles et on sait manipuler les automates pour obtenir les combinaisons de langages. (cf ci-dessus).

... pour faire ceci, on passe, dans un premier temps, de l'automate à un système d'équations. Puis du système d'équations à une expression.

Def : On définit un langage droite par : $L_q^D = \{w \mid q \xrightarrow{w} F\}$. Par analogie, on peut définir les langages gauches $L_q^G = \{w \mid q_0 \xrightarrow{w} q\}$.

Exemples : Prenons l'automate suivant :



On définit donc les langages droites : $L_1^D = a^* b a^* = L$
$L_2^D = a^*$

De même, les langages gauches sont : $L_1^G = a^*$
$L_2^G = a^* b a^* = L$.

On a donc le système d'équations suivants : $\begin{cases} L_1^D = a L_1^D + b L_2^D \\ L_2^D = a L_2^D + \varepsilon \end{cases}$

On résoud ensuite avec l'algorithme de Gauss, en se rappellant que $X = L.X + M \Rightarrow X = L^* M$. $L \not\ni \varepsilon$. (Dans l'autre cas, la formule donne la solution minimale mais non-unique).

On a donc 2 formalismes pour exprimer les langages qui sont les automates et les expressions. Le premier est adapté pour l'algorithmique tandis que le second est mieux pour l'IHM.

Théorème : les pbs suivants sont décidables. $L \neq \emptyset$; $L \cap M = \emptyset$; $L \subset M$; $L = \Sigma^*$. (les algos sont assez triviaux).

Théorème de Myhill-Nerode : Etant donné un langage L (vu comme un ensemble de mots), on veut savoir s'il est régulier et, si oui, trouver un automate.

On définit les ~~quotients~~ (langages droites) pour chaque mot w par : $w \backslash L = \{u \mid w.u \in L\}$. On a l'équivalence définit par $w_1 \simeq w_2$ ssi $w_1 \backslash L = w_2 \backslash L$.

Exemples : Prenons $L = \{a^{3k+1} \mid k \in \mathbb{N}\}$.

On a alors les quotients suivants : $\varepsilon \backslash L = L = \{a^{3k+1}\}$
$a \backslash L = \{a^{3k}\}$
$a^2 \backslash L = \{a^{3k+2}\}$
$a^3 \backslash L = \{a^{3k+1}\}$

On a donc 3 classes d'équivalences, qui seront les états de notre automate. On acceptera dans l'état qui contient le mot vide.

Théorème de Myhill-Nerode : L est un langage régulier ssi il y a un nombre fini de quotients distincts (c'est-à-dire qu'on a un nombre fini de classes d'équivalence).

On peut donc en déduire un automate dont la construction est donné ci-dessus. Cet automate est déterministe et minimale.

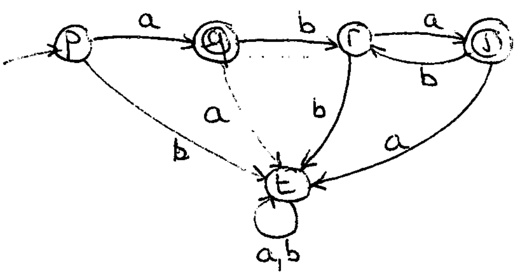
Rem : On utilise plutôt le lemme de pompage pour prouver la non-régularité des langages car il est plus facile à manipuler.

Théorème : Pour chaque automate déterministe A tq $L(A) = L$. On peut obtenir l'automate de Myhill-Nerode en fusionnant les états de L (donc $|M| < |A|$).

Procédure de minimisation : Pour les états de A, on dit que $p \approx q$ ssi $L^D(p) = L^D(q)$. On fusionne tous s états équivalents. Si tout se passe bien, on obtient l'automate de Myhill-Nerode.

m : Tout se passe toujours bien mais on ne fera pas la démonstration ici.

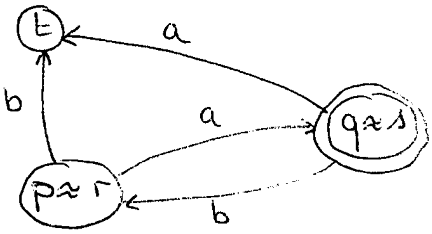
xemple : Prenons l'automate suivant :



Nous aurons donc les langages droites suivant :

$$L_p^D = a(ba)^*$$
$$L_q^D = (ba)^*$$
$$L_r^D = a(ba)^*$$
$$L_s^D = (ba)^*$$

'automate minimale obtenu est donc :



m : - On a pas d'automates non-déterministe minimale (mais un automate non-déterministe peut être eaucoup plus petit que l'automate déterministe minimale).

- Il nous faut un algorithme pour minimiser.

Applications des automates.

On va voir 2 applications des automates : le pattern-matching et le BDD. Dans ces 2 cas, il y a un algo naïf (mauvais), algo automate et un algo utilisé (inspiré de l'algo automate). On insistera essentiellement ~~sur~~ l'algo automate et ~~n~~ verrons comment il a été amélioré pour obtenir l'algo courant.

Le premier pb est celui de la recherche de motif. Il y a 2 manières de l'énoncé : " Etant donné un mot w et une ~~e~~xpression regulière f. Trouver tous les sous-mots w " ou " Etant donné un mot w et un mot u. Trouver toutes ~~les~~ occurences de u dans w ". On va se pencher sur la première manière de l'énoncé. En effet, la seconde peut-être ~~vue~~ comme un cas particulier de l'autre.

~~Un~~e autre manière de le dire : Soit w un mot que l'on peut representé par $u_1 f u_2$. Est-ce que w appartient au ~~lan~~gage. Ou encore : trouver les prefixes de $w \in \{\Sigma^* f\}$.

~~La~~ solution pour répondre à ce pb est de construire un automate A pour $\Sigma^* f$. On donne w à cet automate A. Quand A ~~est~~ dans un état final, on signale une occurrence.

~~La~~ question du déterminisme se pose. Le déterministe est grand, difficile à obtenir, mais facile à appliquer à w ; tandis ~~qu~~e l'autre est facile, petit mais pas simple à appliquer. Nous allons présenter ici les 2 algos qui sont largement ~~uti~~lisés.

```
Algo Det Faire
    De f, on construit A tq  L(A) = [Σ*f];     // |A| = O(m = |f|).
    Determiniser A pour obtenir B;              // |B| = 2^O(m)
    On donne w à B;                             // O(n) pour f fixe.
FAlgoDet
```

~~N~~B : La complexité de chaque instruction est égale à la taille de l'automate obtenu.
~~La~~ complexité totale de cet algo est donc, à peut près, $2^{O(m)} + n.m$ En pratique, $O(n)$ puisqu'on a généralement de ~~pe~~tits automates auquel on passe de long mot. (Ex : une petite exp. reg. donnée à 'grep' sur un enorme fichier).

```
Algo NonDet Faire
    De f, on construit A tq  L(A) = [Σ*f];           // |A| ≤ 2|f|
    Eliminer les ε_transitions pour avoir B;        // |B| = |A|
    On donne w à B;                                  // O(n.|A|²) (peut être un peu plus).
FAlgoNonDet
```

~~L'~~algorithme non-déterministe est donc en $O(n|A|^2)$.
~~R~~q : $O(n.m)$ si on s'y prend bien.
~~On~~ peut donc faire le bilan suivant :

Dét	Non-Dét
Pré-traitement lourd	Pré-traitement simple
Traitement facile	Traitement lourd
Espace mem : 2^m	Espace mem : m (ou m^2)
Algo lin. en O(n)	Algo itératif (une boucle pour chaque lettre de w).

yons maintenant comment trouver les occurrences d'un mot u donné dans un mot w donné. Pour resoudre ce pb, on plein d'algos. Commençons par présenter l'algo naïf :

```
Algo Naïf
|   Pour i de 0 à n-m-1 Faire
|   |
|   |      Comparer u avec w_i ... w_{i-m-1}
|   Fpour
FAlgo.
```

t algorithme a une complexité de $O((n-m).m)$

algos intelligents sont aussi très nombreux. Notamment celui de Knuth-Morris-Pratt donné par :

```
Algo K.M.P Faire
|   Construire l'automate deterministe minimal pour Σ*u A ; // la construction est donnée plus loin
|   Appliquer w à A ;
FAlgo KMP.
```

idée sur lequel repose cet algorithme est que l'automate minimale mémorise le plus grand préfixe de u qui est le ffixe du mot déjà lu.

appellons l'équivalence de Nerode : $w_1 \approx w_2$ (i.e. w_1 et w_2 mènent au m̂ état de l'automate minimal).

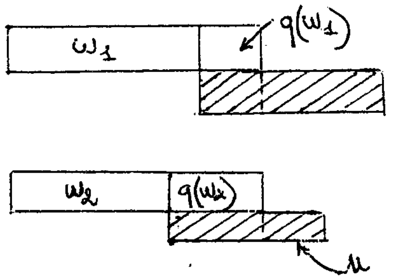
$$\Updownarrow$$

$$\forall y \quad w_1 y \in \Sigma^* u \Leftrightarrow w_2 y \in \Sigma^* u.$$

oposition : On définit $q(w) = \left\{ \text{le + gd } X \;\middle|\; \begin{array}{l} X : \text{préfixe de } u \\ X : \text{suffixe de } w \end{array} \right\}$,

Alors $w_1 \approx w_2 \Leftrightarrow q(w_1) = q(w_2)$.

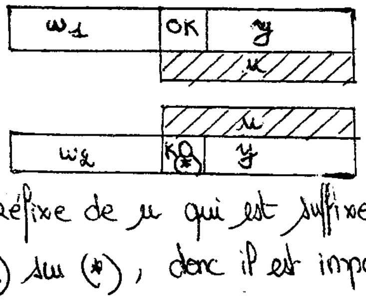
euve : . Supposons que $q(w_1) \neq q(w_2)$. Sans pertes de généralités, nous poserons $|q(w_2)| > |q(w_1)|$.



$u = q(w_2).y$,

Pour cet y, on a $w_2 y$ se termine par u, et $w_1 y$ ne se termine pas par u.
Donc $w_1 \not\approx w_2$.

. Supposons que $w_1 \not\approx w_2$,
Pour un y donné, $w_1 y \in \Sigma^* u$ et $w_2 y \in \Sigma^* u$



onc il y a un préfixe de u qui est suffixe de w_1 et non de w_2. On considère maintenant $q(w_1)$. Il est est fférent de $q(w_2)$ sur (*), donc il est impossible que $q(w_1) = q(w_2)$.

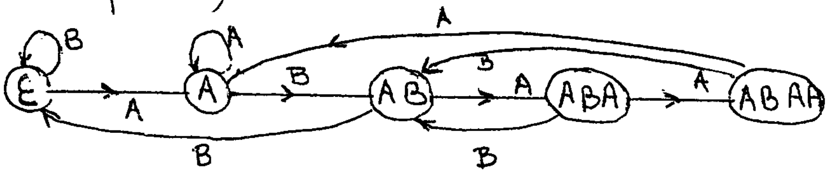
n définit l'automate minimale pour $\Sigma^* u$ par $Q = \{\text{préfixe de } u\}$

$q_0 = \langle \varepsilon \rangle$

$\Gamma = \{<u>\}$

$\delta(<x>, a) = \{ <va>$ si 'va' préfixe de u
le ⊕ lg suffixe de 'va', préfixe de u sinon $\}$

emple : Voici l'automate pour le mot $u = ABAA$.

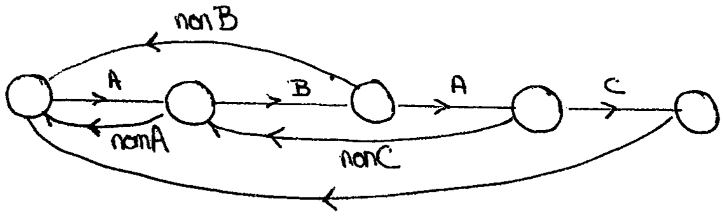


n algorithme pour résoudre notre pb pourrait donc être :

Algo Faire
| Construire cet aut. $(m+1)$ états, dét, complet ;
| Appliquer à w ;
FAlgo.

éanmoins, l'inconvénient est que si Σ est grand, l'automate a beaucoup de transitions. Il peut donc être long construire.

oyons maintenant comment construire l'automate de KMP qui permet d'appliquer l'algorithme. On verra un emple pour le mot ABAC :



a construction de cet automate a une complexité de $O(m)$. Le calcul pourra être fait en $O(n+m)$.

Nous allons maintenant voir comment traiter le second pb (i.e la BDD) avec des automates. Là encore, nous verrons algorithme automates dans un premier temps puis comment il a été amélioré pour obtenir l'algorithme courant. Le problème la manipulation des fonctions booléennes : $f : \{0,1\}^n \longrightarrow \{0,1\}$. On connait les formules propositionnelles telles que, exemple, $f(x,y,z) = ((\overline{x \vee y}) \Rightarrow z) \vee x$; les formules propositionnelles restreintes (en forme normale conjonctive ou bien forme normale disjonctive). On connait aussi les tables de vérité mais on ne sait pas comment les faire sur n variables. a aussi les ensembles énumérées comme par exemple $\{(110), (111), (100)\}$.

problèmes algorithmes qui se posent sont la réalisation, facilement, des opérations $\vee, \wedge, \Rightarrow, \ldots$; la réponse à la tisfaisabilité d'une formule ($\exists x \; f(x) = 1$?) ; l'équivalence de 2 formules.

s applications peuvent être les circuits électroniques, notamment dans le cas de l'équivalence de 2 formules (spec $\Leftrightarrow$ calcul alisé par le circuit). On peut aussi tester ces mêmes circuits. Par exemple, un circuit réalise f normalement et g lorsqu'il est panne ; il faut trouver un x pour lequel $f(x) \neq g(x)$ et on le met en entrée afin de connaître l'état du circuit. e troisième application pourrait être le Model-Checking.

: . Le problème de satisfaibilité d'une formule est un pb NP-complet.
. L'équivalence de 2 formules est un pb co-NP-complet.
. Il y a 2^{2^n} fonctions booléennes à n variables. Les représenter nécessitera donc 2^n bits.

ns le table suivant, on juge la difficulté d'une opération selon la représentation qu'elle a :

	Formules	CNF	DNF (listes)
Calcul de F(X)	Facile	Facile	Facile

	Formules	CNF	DNF (listes)
$F,g \mapsto f \vee g$	$m+n+1$	mn	$\leq m+n+1$
sat	NP_complet	NP_complet	facile
Représentat° d'une Formule de taille n	Triviale	exp.	exp.
$f \Leftrightarrow g$	co_NP_complet	co_NP_complet	co_NP_complet

n peut donc en conclure que ça ne permet pas de répondre facilement au pb. Nous allons donc trouver 2 autres structures de données qui sont assez efficaces dans les cas pratiques. Il subsiste néanmoins 2 grandes limitations : pour certaines ctions, on a une taille de 2^n et ça ne donne pas de solutions polynomiales pour sat, $f \Leftrightarrow g$, ...

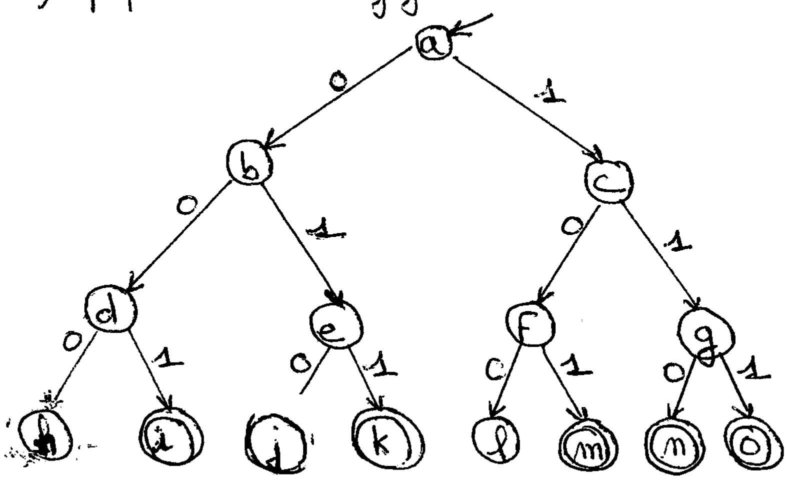
me première structure de données que l'on va étudier est un automate, $A = \langle \Sigma, L \rangle$ avec $\Sigma : \{0,1\}$
$L : \{x_1 \dots x_n \mid f(x_1, \dots x_n) = 1\}$.

langage de cet automate est fini mais grand. Notre automate A qui reconnait L sera déterministe (et minimale).

: Prenons $f(x, y, z) = (x \wedge y) \vee z$.

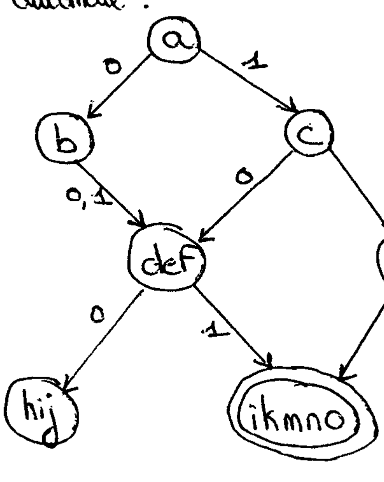
Donc $L_f = \{110, 001, 011, 101, 111\}$.

L'automate naïf qui reconnait ce langage est donc :



arbre décision binaire qui a toujours une taille 2^n (et pas seulement dans certains cas).

On peut minimiser cet automate :

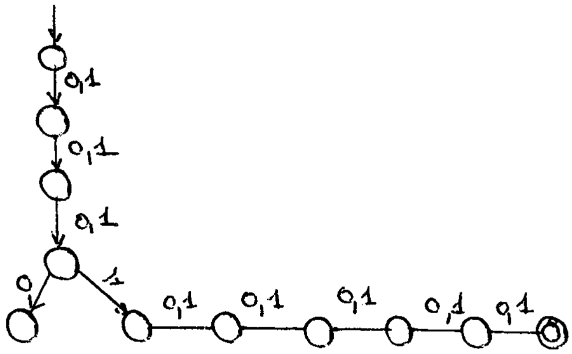


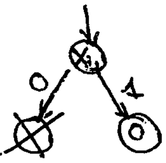
rq : L'automate précédent a plusieurs propriétés : - il est acyclique.
- la minimisation est simple (1 parcours de bas en haut).
- il peut être construit inductivement.

- Pour $\neg x_i$, il suffit d'inverser l'état inverseur et l'état accepteur dans l'automate donné
écédemment (il est toujours possible de repousser les négations au niveau des variables).
- Minimiser avec un parcours de bas en haut (cf TD pour plus précision).

us allons voir maintenant comment régler nos problèmes posés au départ avec une telle structure de données :
- SAT : vérifier l'existence d'un état final (mais la construction peut être exponentielle).
- $f \Leftrightarrow g$: vérifier que les automates minimaux sont équivalents (facile).

tte structure de données a néanmoins un inconvénient : l'automate teste toutes les variables. Par exemple, pour un
t de longueur 10, l'automate sera

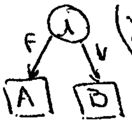


ne optimisation simple serait : 
. Mais on n'obtient pas un automate mais un BDD.

f : Un BDD (Binary Decision Diagramm) sur les variables $x_1, \dots, x_m$ est un graphe acyclique où chaque
ud n a un indice $ind(n) \in \{1, \dots, m\}$; Chaque feuille est étiquetée par 0 ou 1 ; chaque nœud non-feuille
2 fils ($f(n)$ et $v(n)$) ; les indices de $f(n)$ et $v(n)$ sont supérieurs à $ind(n)$; il y a une seule racine.

f : La fonction calculée par un BDD sur $X_1 \dots X_n$ peut être définie inductivement :

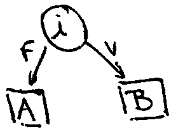
<u>cas de base</u> :
- $\boxed{1}(X_1 \dots X_n) = \boxed{1}$
- $\boxed{0}(X_1 \dots X_n) = \boxed{0}$

<u>cas inductif</u> : 
$(X_i \dots X_n) = \begin{cases} A(X_{i+1} \dots X_n) & \text{si } X_i = 0 \\ B(X_{i+1} \dots X_n) & \text{si } X_i = 1. \end{cases}$

<u>rme</u> : On va donner la définition inductive d'un BDD sur $X_1 \dots X_n$ dont on s'est servi dans la définition précé-
te. : On a 2 cas de base et un cas inductif donnés ci-dessous :

<u>cas de base</u> : $\boxed{0}$

$\boxed{1}$

<u>cas inductif</u> : 
où A et B sont des BDD sur $\{X_{i+1} \dots X_n\}$.

<u>éorème</u> : Pour chaque feuille, il existe un BDD qui la calcule.

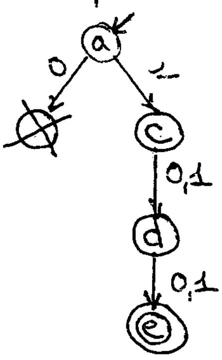
<u>monstration</u> : Soit l'arbre binaire complet de profondeur n. On a donc 2^n feuilles. Dans la feuille, à l'adresse
$\dots X_n$, on écrit $f(X_0, \dots, X_n)$.

<u>ation</u> : Pour un nœud n, soit $F_n(X_1 \dots X_n)$, la fonction booléenne calculée par la sous-BDD avec la racine n.
e dépend uniquement de $\{X_{ind(n)} \dots X_m\}$.

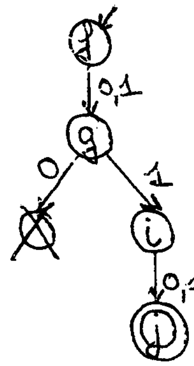
f : Un BDD est réduit (appellé ROBDD) est un BDD tq :
- $\forall n \; f(n) \neq v(n)$
- $\forall n_1 \neq n_2, \; F_{n_1} \neq F_{n_2}$

nstruisons l'automate précédent de manière inductive :

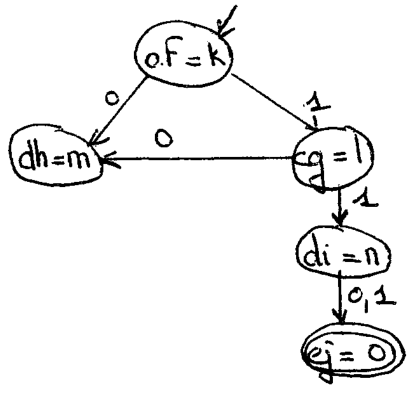
A(x)



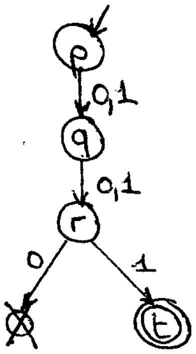
A(y)



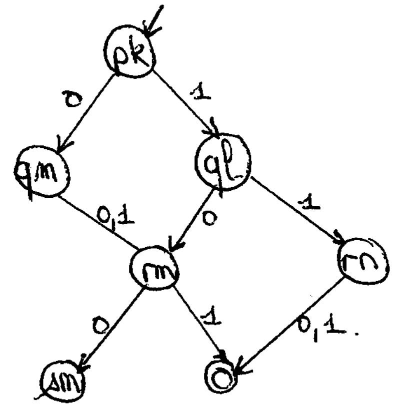
A(x∧y)



A(z)

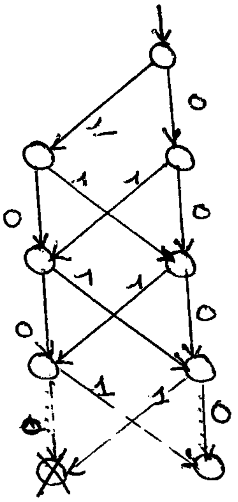


A((x∧y)∨z)



q: les minimisations sont ultra-simples avec cette méthode.

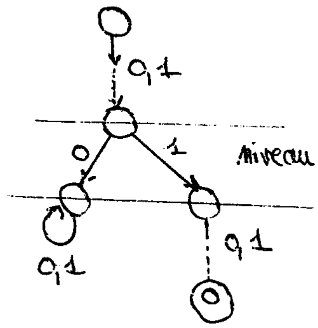
xemple : Prenons maintenant $f(x_1, \dots, x_n) = (x_1 + \dots + x_n) \bmod 2$.
$= x_1 \oplus x_2 \cdots \oplus x_n$. Cette fonction est difficile à exprimer avec les connecteurs assiques $(\vee, \wedge, \neg, \dots)$. Par contre, l'automate est :



Notre automate a 2^n états.

yons maintenant comment construire ces automates formellement :

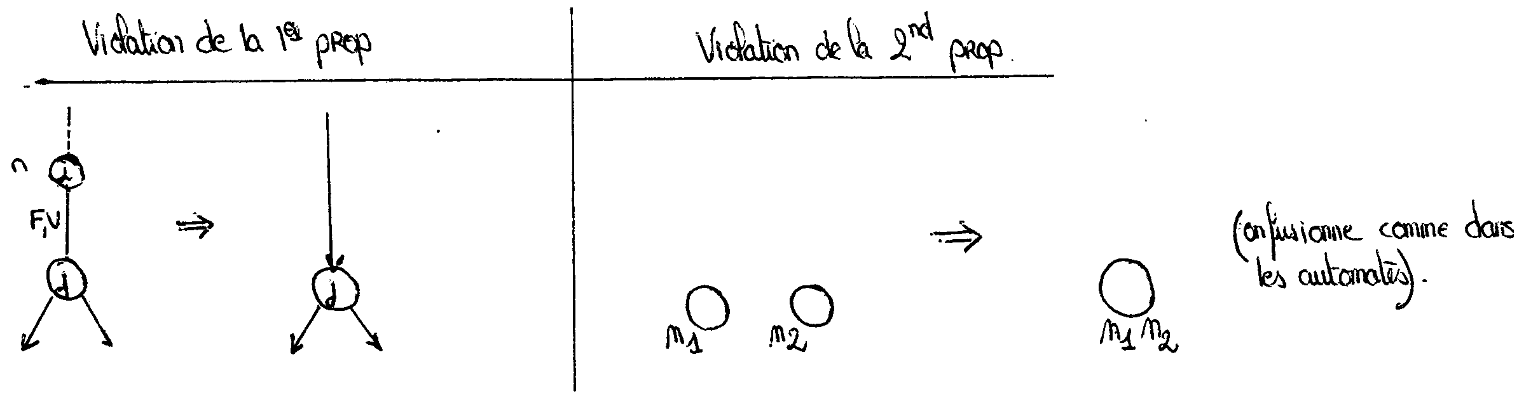
- Pour la variable x_i, on construit l'automate suivant :



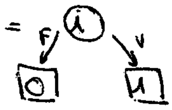
- Pour $f \wedge g$ et $f \vee g$, on construit l'automate produit et on minimise.
- Pour $\neg f$ c'est désagréable, on ne le fera pas ici.

(Pr)oposition : Chaque fonction peut-être calculée par un BDD réduit. L'algo de réduction sera vu en TD.

(Ex)emple : Nous allons voir 2 exemples de réductions.



(Th)éorème : le BDD réduit est unique. Donc si 2 BDD réduits calculent la même fonction, alors ils sont (is)omorphes.

(Sc)héma d'utilisation : - Construire un BDD(x_i) = 

- Si on a un BDD pour f, alors pour avoir $\neg f$, il suffit de changer les étiquettes des feuilles.

- Soient un BDD pour f et g, alors $(f \wedge g)$ et $(f \vee g)$ s'obtiennent en faisant le produit des BDD, puis en réduisant ce nouveau BDD.

- SAT(f) s'obtient en vérifiant qu'aucune feuille n'est étiqueté $\boxed{0}$.

- $(f \Leftrightarrow g)$ nécessite que f et g aient le même BDD réduit.

Voyons maintenant un troisième problème que l'on va résoudre en utilisant les automates : la procédure de décisions (de) l'arithmétique de Presburger. C'est un pb largement utilisé dans les algorithmes de vérification de programmes. Dans (un) premier temps, nous allons revoir l'isomorphisme des expressions régulières.

(Ra)ppel : Homomorphisme lettre-à-lettre.

Soit $f : \Sigma_1 \to \Sigma_2$.

. Etant donné f et $w \in \Sigma_1^*$ (avec $w = w_1 \cdots w_n$),
Alors on définit $f(w) = f(w_1)f(w_2) \cdots f(w_n) \in \Sigma_2^*$.

. Etant donné f et un langage $L \subset \Sigma_1^*$,
Alors $f(L) = \{f(w) \mid w \in L\} \subset \Sigma_2^*$.

. Etant donné f et $M \subset \Sigma_2^*$,
Alors $f^{-1}(M) = \{w \mid f(w) \in M\} \subset \Sigma_1^*$.

(Ex)emple : Soit $f : \begin{cases} a \mapsto c \\ b \mapsto c \end{cases}$ avec $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{c\}$.

Donc $f(abba) = cccc$
$f((ab)^*) = (cc)^*$
$f^{-1}(cc) = (a+b)(a+b)$.

(Th)éorème : Soit f un homomorphisme lettre-à-lettre,
Si f est régulier, alors $f(L)$ est régulier
Si M est régulier, alors $f^{-1}(M)$ l'est aussi.

(Id)ée de démonstration : Soit L un langage régulier
$L = L(A)$ avec $A = \langle Q, \Sigma, q_0, F, \Delta \rangle$.

. On construit l'automate B définit par $B = \langle Q, \Sigma_2, q_0, F, \Delta' \rangle$ avec $\Delta' = \{ p \xrightarrow{f(a)} q \mid p \xrightarrow{a} q \in \Delta \}$.

Il nous reste à montrer que le calcul de l'un implique le calcul de l'autre et vice-versa : aucune difficulté.

Pour la seconde partie du théorème, l'idée est la même, on construit un automate pour M, langage régulier définit par $M = \langle Q, \Sigma_2, q_0, F, \Delta \rangle$. On construit ensuite l'automate $A = \langle Q, \Sigma_1, q_0, F, \Delta' \rangle$ avec $\Delta' = \{ p \xrightarrow{a} q \mid p \xrightarrow{f(a)} q \in \Delta \}$. Là-encore, il nous faudrait démontrer la correction de cette construction.

Exemple : Logique du 1^{er} ordre

- $\forall y \exists x \, (x + x = y)$: fausse sur $\mathbb{N}$, $\mathbb{R}$
  vraie sur $\mathbb{C}$, $\mathbb{B} = \{0,1\}$.
- $\exists x \, (x + x = y)$ sur $\mathbb{N}$ définit $\{ y \text{ pair} \mid y \in \mathbb{N} \}$.

Pour résoudre notre pb, on veut un algorithme qui, pour chaque formule sans variables libres nous dit si elle est vraie ou fausse sur $\mathbb{N}$. Mais un tel algorithme n'existe pas.

Théorème : Pour l'arithmétique de Peano, il n'existe pas d'algorithme de décisions. (Tarski).

Nous allons maintenant travailler sur la théorie de Presburger qui a, comme signature : $0, 1, +, =$ et qui s'exprime sur $\mathbb{N}$.

Théorème : L'arithmétique de Presburger est décidable, i-e, il existe un algorithme qui, pour chaque formule close de l'arithmétique dit si elle est vraie ou fausse sur $\mathbb{N}$.

Démonstration : Nous allons utiliser les automates pour prouver le théorème ci-dessus. Cette démonstration a été attribuée par Buchi. Dans un premier temps, redéfinissons la formule :

$$\text{Terme} ::= 0 \mid 1 \mid x \mid \text{Terme} + \text{Terme}$$

$$\begin{aligned} \text{Formule} ::= \; & \text{Terme} = \text{Terme} \mid \\ & \neg \text{Formule} \mid \\ & \text{Formule} \vee \text{Formule} \mid \\ & \text{Formule} \wedge \text{Formule} \mid \\ & \forall x \, (\text{Formule}) \mid \\ & \exists x \, (\text{Formule}). \end{aligned}$$

Une valuation v sur un ensemble fini de variable E est définit par : $v : E \to \mathbb{N}$. Elle donne la valeur de la variable. On a donc : Soit t un terme $E \supseteq VL(t)$,

Et $v : E \to \mathbb{N}$ une valuation.

Alors la valeur de t dans $\mathbb{N}$ est donnée par

$$[0] = 0$$
$$[1] = 1$$
$$[x] = v(x)$$
$$[(t_1 + t_2)] = [t_1] + [t_2].$$

Pour f une formule,

Et la même valuation v,

Alors la valeur de f dans $\{0,1\}$, par $[f]_v$ est définie par induction (trivial).

On notera $[\![f]\!]$, pour toutes formules f non-closes, et pour $E \supseteq VL(f)$, toutes les valuations qui rendent f vraie.

Exemple : Prenons f : $\exists t \, ((x+y)+t) = z$

Et $E = x, y, z$

Alors $[\![f]\!] = \{ v \mid v(x) + v(y) \leq v(z) \}$.

On notera la valuation sous la forme d'un vecteur $\begin{pmatrix} v(x) \\ v(y) \end{pmatrix}$ ou même $\begin{pmatrix} x \\ y \end{pmatrix}$ par abus d'écriture.

idée de la démonstration repose sur, pour chaque formule f, la construction d'un automate qui reconnait $[\![f]\!]$.
va voir comment faire étape par étape. Le premier pas consiste à se débarasser des symboles de fonctions, à savoir ici, en le remplaçant par un prédicat $P(x,y,z)$ définit par $P(x,y,z) \Leftrightarrow x+y=z$.

me : Pour chaque formule, il existe une formule équivalente sans + mais avec P. Il suffit de le faire pour les mules $t_1 = t_2$. On le fera donc par récurrence sur le nombre de ~~+~~ dans t_1 et t_2.

O : pas de +, ok.

n+1 : soit t_1, soit t_2 contient un +. Considerons, sans pertes de généralités qu'il s'agit de t_1.
a donc $t_1 \cdot tg+td$, d'où $tg+td = t_2 \Leftrightarrow \exists x,y,z\ (P(x,y,z) \wedge x=tg \wedge y=td \wedge z=t2$. En pliquant l'hypothèse de récurrence à $(x=tg)$, $(y=td)$ et $(z=t2)$, on obtient une formule sans +.
à fin de cette première étape, on a donc une nouvelle logique que l'on peut définir inductivement par :

Terme ::= 0 | 1 | x
Formule ::= Terme = Terme |
P(Terme, Terme, Terme) |
Formule A Formule avec $A = \{\wedge, \vee, \neg\}$.

codage de valuation : Soit $E = \{x_1, \dots, x_k\}$ un ensemble fini de variables.
Soit $v : E \to \mathbb{N}$ une valuation.
Soit $\Sigma = \{0,1\}^k$,
Alors le mot w en Σ^* sera de la forme

$v(x_1)$ "valuation de x_1 en Little Endian"
$v(x_2)$ ⋮
⋮
$v(x_n)$ $\underbrace{}_{1^{\text{re}} \text{ lettre de } w}$.

emple : $E = \{x, y, z\}$
Avec la valuation $x \mapsto 5$
$y \mapsto 3$
$z \mapsto 7$
Donc le mot w sera :

1	0	1
1	1	0
1	1	1

$1^{\text{ère}}$ lettre.

dit que w représente v si $\forall i$, $V(x_i) = \sum_{m=1}^{l} w_m^i \cdot 2^{m-1}$.

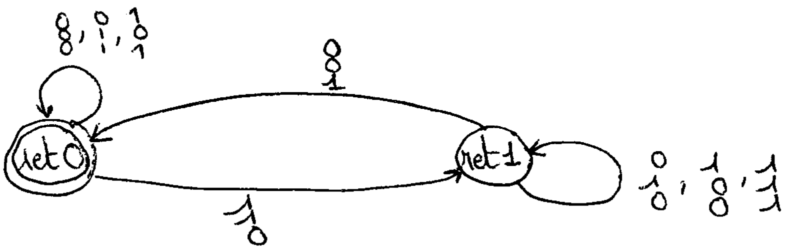
me : Pour chaque formule f et chaque ensemble $E \supset VL(f)$, l'ensemble de w tels que $v = [w]$ rend la formule f vraie est régulier. On peut donc construire un automate de manière algorithmique.

emple : . $f : x = 1$ $E = \{x\}$
Donc $L = \{w \mid w \text{ est syst. binaire Little Endian}\} = 10^*$.
D'où l'automate 

· $g : P(x,y,z)$ $E = \{x,y,z\}$.

· Donc $L = \left\{ \begin{smallmatrix} x \\ y \\ z \end{smallmatrix} \;\middle|\; x + y = z \right\}$.

D'où l'automate sera :



...n a les relations suivantes entre les différentes représentations :

$$\begin{tikzcd} \text{Formule } f,\ E \ \ E \supseteq FV(f) \arrow[r, "\text{construct}^{\circ}", "\text{inductive}"'] \arrow[d, dash, "\text{sémantique}"'] & \text{un automate } A \arrow[d, "\text{langage de l'automate}"] \\ \text{un ens de valuation} & \text{un ens. de mots} \arrow[l, "\text{codage}"] \end{tikzcd}$$

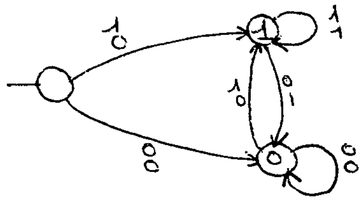
...n construit un automate A à partir de f et E en utilisant l'induction structurelle sur f. La proposition à assurer ...t la suivante : $L(A) = c^{-1} [\![f]\!]_E$

...q : $[\![f]\!]_E$ représentent toutes les valuations qui rendent f vraies sur E.

. c est la fonction de codage. Donc c^{-1} donne tous les mots sur $[0,1]^*$ qui code ces valuations.

...se : Soit f une formule f atomique et E quelconque,

Abrs
$0 = 1 \longrightarrow \bigcirc$
$0 = 0 \longrightarrow \circledcirc\, \varepsilon$
$1 = 1 \longrightarrow \circledcirc\, \varepsilon$
$X = 0 \quad E = \{x\} \quad L = 0^*$ (facile à construire)
$X = 1 \quad E = \{x\} \quad L = 10^*$ (facile à construire également)
$X = Y \quad E = \{x, y\} \quad L = \left(\begin{smallmatrix}1\\1\end{smallmatrix} + \begin{smallmatrix}0\\0\end{smallmatrix}\right)^*$
$P(x, y, z) \quad E = \{x, y, z\}$: on a déjà construit cet automate précédemment.
$P(x, x, z) \quad E = \{x, y\}$, l'automate est donc :
⋮



(rmq : l'état donne le dernier x lu).

...mq. Pour que la base soit correcte, il faut déterminer pour un ens. E quelconque (qui contient néanmoins les variables ...ores de f) ce qui n'est pas le cas ici. Il faudrait modifier un peu.

...mme : Si on a un automate A pour f, E et $E' \to E$,
Alors on peut construire un automate pour f, E'.

...ici, on peut donc rajouter des variables inutiles pour avoir un E quelconque.

...uve : Soit $E \subset E' = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ y_1 \\ \vdots \\ y_k \end{pmatrix}$

Par hypothèse y_i n'est pas dans $FV(f)$

Soit $h : E' \to E$

Alors $w \in \{0,1\}^{k+m}$, $w \in c^{-1}([f]_{E'}) \Leftrightarrow c(w) \leftarrow [\![f]\!]_{E'} \Leftrightarrow h(c(w)) \in [h]_E$

$\Rightarrow c(h(w)) \in [\![f]\!]_E \Leftrightarrow h(w) \in L(A))$.

On peut vérifier maintenant, au cas par cas, que $\forall f$ atomique, tout $E \supset VF(f)$, on a un automate.

Induction : On suppose que cela est prouvé pour f et g.

On a donc, pour tous $E \supset VF(f) \cup VF(g)$

On a donc : * $\neg f$: on a un automate A pour les mots w tq $c(w) \in [\![f]\!]_E$. Pour avoir tous les mots tq $c(w) \in [\![\neg f]\!]_E$, on construit l'automate pour le complément.

* $f \vee g$: on construit l'automate comme l'union des automates f et g.

* $\exists$: on a un automate pour $f(x, \vec{y})$ avec $E = \{x, \vec{y}\}$. On a aussi un automate pour avec $E = \{x, \vec{y}\}$. On veut donc un automate qui accepte $u = \vec{y}$ ssi $\exists x$ $f(x, \vec{y})$ (un homomorphisme lettre-à-lettre)

Rq : • Pour les E plus larges que $\vec{y}$, on applique le lemme énoncé précédemment.

• On a des outils qui construisent ces automates automatiquement.

Théorème : L'arithmétique de Presburger est décidable.

Preuve : Soit f une formule close

Donc $FV = \emptyset$

Par le lemme principal, on construit un automate A pour f avec $E = \{x\}$ (x quelconque),

Alors $w \in L(A)$ ssi pour la valuation $x \mapsto c(w)$, on a f ssi f est vrai.

Donc $L(A) = \begin{cases} \{0,1\}^* & \text{si } f \text{ est vraie} \\ \emptyset & \text{si } f \text{ est fausse} \end{cases}$

On distingue les 2 cas en testant A pour le langage vide.

On a donc un algorithme de décisions qui est : • construire A pour f, $\{x\}$

• Si $L(A) = \emptyset$, alors retourner faux

Sinon retourner vrai.

Les langages réguliers en logique.

On verra, dans ce chapitre, comment définir les langages réguliers en logique. On possède déjà quelques ils qui sont : les automates et les expressions régulières. Les automates sont plutôt utilisés pour des représenta-ns machine ou mathématiques tandis que les expressions se retrouvent dans les interfaces "hackers". Mais il exi-e aussi beaucoup d'autres formalismes comme les modèles logiques (MSO - équivalentes aux automates - ; FO et L)

<u>gique du 1er Ordre pour déf. langage</u> : on va donner la signature et un modèle pour cette théorie.

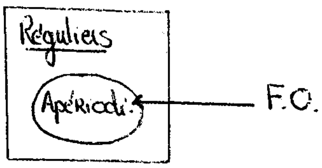
* <u>Signature</u> : $<$, $=$, $P_a(x)$ pour tous dans Σ.
* Modèle qui correspond au mot $w = w_1 --- w_n$ peut être défini par :
  - Dom $\Rightarrow \{1, 2, \dots\}$
  - $=$, $<$: habituelles
  - $P_a(i)$ est vrai $\Leftrightarrow w_i = a$.

n notera ce modèle $M(w)$

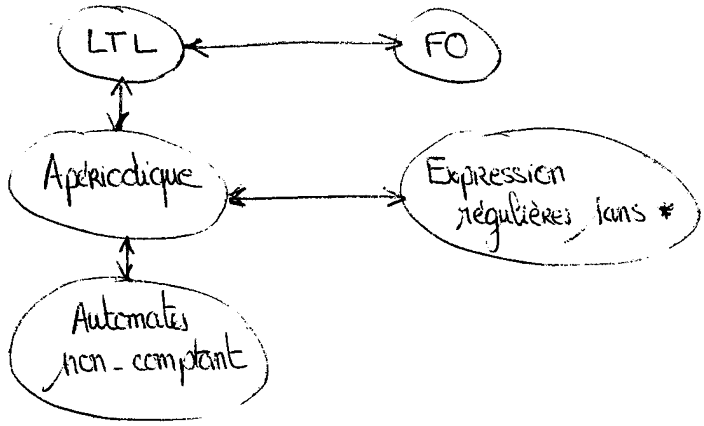
pour le modèle $M(w)$, la formule f est vraie, on dit que w satisfait f ($w \models f$). On pose $L(f) = \{w \mid w \models f\}$

<u>emple</u> :
- $\forall x\ P_b(x)$ définit le langage b^*
- Pour définir $\Sigma^* a \Sigma^*$, on a : $\exists x \exists y\ P_a(x) \wedge P_b(y) \wedge y > x \wedge\ !\exists z\ (z > x \wedge z < y)$
- $(ab)^* = \overline{\Sigma^* aa \Sigma^*} \cap \overline{\Sigma^* bb \Sigma^*} \cap a\Sigma^* \cap \Sigma^* b$
- $(aa)^*$ est, quant à lui, impossible à définir dans une logique du 1er ordre.

n a la situation suivante :



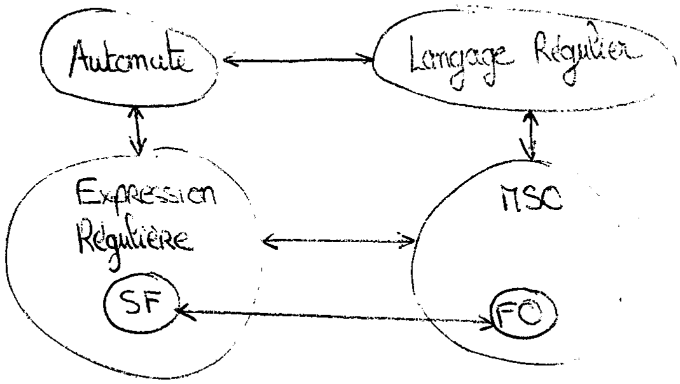
<u>quivalence</u> :



<u>f</u> : Une expression régulière sans * (RSF ou SF) $= \emptyset \mid a \mid \varepsilon \mid RSF + RSF \mid RSF.RSF \mid \overline{RSF} \mid \Sigma^*$.
. Un langage est sans étoile ssi peut être défini par une RSF.

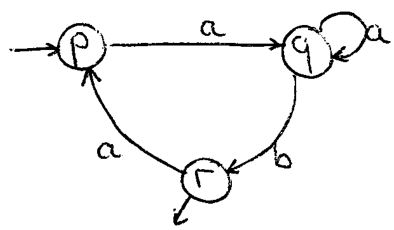
<u>xemple</u> : $(ab)^*$ est un langage sans étoile puisqu'on peut le définir par . $\overline{\Sigma^* aa \Sigma^* + \Sigma^* bb \Sigma^* + b\Sigma^* + \Sigma^* a}$

us généralement, on a le schéma d'équivalence qui suit :



Nous allons maintenant établir l'équivalence entre la logique MSO (Monadique du Second Ordre) et les automates.

Soit un automate comme suit :



(A)

On décrit le langage de A en logique (que l'on a pas encore décrite ni définie). On va donc définir les 3 prédicats suivants : $P(i)$: l'automate à la position i est dans p.
$Q(i)$: l'automate à la position i est dans q.
$R(i)$: l'automate à la position i est dans r.

On aura aussi $a(i) \Leftrightarrow w_i = a$ et réciproquement $b(i) \Leftrightarrow w_i = b$.

On a donc l'expression suivante : $P(0)$ ⟶ définit l'état initial.

$$\left.\begin{aligned} \wedge \forall i,j \text{ tq } j=i+1 \ [&(P(i) \wedge a(i) \wedge Q(j)) \\ &\vee (Q(i) \wedge a(i) \wedge Q(j)) \\ &\vee (Q(i) \wedge b(i) \wedge R(j)) \\ &\vee (R(i) \wedge a(i) \wedge P(j))] \end{aligned}\right\} \text{ permet de respecter } \Delta.$$

rmq : "$j = i+1$" devrait être redéfini en logique pour être tout à fait rigoureux.

On a donc l'équivalence suivante : $w \in L(A) \Leftrightarrow \exists P, Q, R$ tq $\forall x$

$$\left.\begin{aligned} (&\neg(P(x) \wedge Q(x)) \\ &\wedge \neg(Q(x) \wedge R(x)) \\ &\wedge \neg(P(x) \wedge R(x)) \end{aligned}\right\} \text{ qui assure l'exclusion mutuelle}$$

$\wedge \exists k \, (R(z) \wedge \neg \exists z \, (k < z))$ ⟶ qui définit l'état final.

$\wedge$ l'expression précédente

Déf : La logique monadique de 2^{nd} ordre (MSO) avec la signature : $0, >, \Sigma$.

On a donc : Terme $:= 0 \mid x$

Formule $:= t_1 = t_2 \mid t_1 < t_2 \mid X(x) \mid a(x) \mid For \vee For \mid \neg For \mid \exists x \, For \mid \exists X \, For$.

rmq : . X est un ens. de variables tandis que x est un entier, une variable.

. $X(x)$ se lit : "x satisfait X".

. Il y a autant de formules $a(x)$ que de lettres dans Σ.

Exemple : . $\exists P \ \forall x \ \neg P(x)$

. $\forall P \ (P(0) \wedge \forall x, y \ ((y = x+1) \wedge P(x) \rightarrow P(y)) \rightarrow \forall z \ P(z)$. On peut remarquer qu'il s'agit de l'axiome de l'induction.

Rmq : • Monadique signifie que tous les prédicats ont au plus une variable.
• 2^{nd} ordre : les prédicats peuvent être variables

Un modèle qui correspond à un mot $w_1 \dots w_{n-1} \in \Sigma^*$. Son domaine $= \{0, \dots, n\}$. On a : $a(i)$ vrai $\Leftrightarrow w_i = a$
Et $(i<j)$ et $(i=j)$ gardent leurs interprétations standards

Une valuation d'un ens. de variables est donné par :
$$\begin{pmatrix} x_1 \\ \vdots \\ x_p \\ X_1 \\ \vdots \\ X_q \end{pmatrix} = \begin{pmatrix} v(x_1) \in \mathcal{D} \\ \vdots \\ \\ \vdots \\ \\ v(X_q) \subseteq \mathcal{D} \end{pmatrix}$$

Il nous reste à définir f avec les variables libres $x_1 \dots x_p X_1 \dots X_q$ qui est vraie sur un mot w pour une valuation v : $v \models f$.

Def inductive : $\cdot\ x < y \Leftrightarrow v(x) < v(y)$
$\cdot\ a(x) \Leftrightarrow w_{v(x)} = a$
$\cdot\ X(x) \Leftrightarrow v(x) \in v(X)$.

Notation : Soit une formule MSO f close,
Alors $S(f) = \{w \mid v \models f\}$.

Théorème : (1). Pour chaque formule f, $S(f)$ est régulier.
(2). Pour chaque langage régulier, il existe une formule f tq $L = S(f)$.

Preuves : (2) - cette preuve a déjà été précédemment.
(1) - Pour cela, on doit construire un automate qui reconnaît le mot et toutes les valuations des variables

Codage : On va coder les mots et leur valuation.
Soit $w = w_0 \dots w_{n-1}$ un mot
$v(x) = s$
$v(X) = M$.

mot

w_0	w_1	-----	w_s	-----	w_{n-1}	$\circ$
0	0		1		0	0
1	1	0	1		1	1
$\in M$	$\in M$		$\in M$		$\in M$	$\in M$

Lemme principal : Soit $f(x_1 \dots x_p X_1 \dots X_q)$ une formule $S(f) = \{ \begin{smallmatrix} w \\ x \\ X \end{smallmatrix} \mid f$ est vraie $\}$,
Alors $S(f)$ est régulier.

Démonstration : On va faire une induction structurelle pour montrer cela.
Cas de base : Soit Σ l'alphabet de w, $A = \Sigma \times \{0,1\}^{p+q}$.
$\cdot\ x<y \quad : (A^* . x . A^* . y . A^*) \cap k$ où k : tous les mots $\begin{bmatrix} w \\ x \\ X \end{bmatrix}$ tq un seul "1" par piste x.
$\cdot\ a(x) \quad : (A^* . \begin{bmatrix} a \\ 1 \\ * \\ * \end{bmatrix} . A^*) \cap k$.
$\cdot\ X(x) \quad : (A^* . \begin{bmatrix} * \\ 1 \\ * \\ 1 \end{bmatrix} . A^*) \cap k$.

Cas inductifs : • Si $S(f)$ régulier, $\Rightarrow S(\neg f) = \overline{S(f)}$ est régulier.
• Si $S(f)$ et $S(g)$ réguliers $\Rightarrow S(f \vee g) = S(f) \cup S(g)$ régulier.
• Si $S(f)$ régulier $\Rightarrow S(\exists x f) = \pi(S(f))$ où π efface la piste x est régulier
• Si $S(f)$ régulier $\Rightarrow S(\exists X f) = \pi(S(f))$ avec le même π

Donc, on a bien démontré (1)

En conclusion, nous avons donc bien démontré l'équivalence : MSO $\Leftrightarrow$ automates.

On va maintenant passer au premier ordre.

Théorème : (1) Soit f une formule close du premier ordre.
Alors $S(f)$ est un langage régulier sans $*$.
(2) Soit L un langage régulier sans $*$,
Alors il existe une formule f du 1er ordre tq $S(f) = L$.

Démonstration : (2). Pour prouver cela, il nous faut un lemme.

Lemme : Soit f une formule,
On peut construire une formule $f|_{x,y}$ tq $f|_{x,y} \models w \Leftrightarrow f \models w$ sur le segment de x à y.
avec $f|_{x,y}$ donnant la restriction de f $w_x \cdots x_y$.

Dem du lemme : . Pour f atomique, $f|_{x,y} = f$,

$\cdot (\neg f)|_{x,y} = \neg(f|_{x,y})$
$\cdot (f \vee g)|_{x,y} = f|_{x,y} \vee g|_{x,y}$
$\cdot (\exists z f)|_{x,y} = \exists z (x \leq z \leq y \wedge f(z))$.

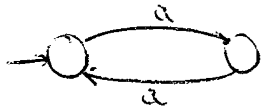
Une preuve inductive bien construite serait mieux !

On a donc les équivalentes suivantes : . $\emptyset \longrightarrow \neg 0 = 0$
. $\Sigma^* \longrightarrow 0 = 0$
. $\overline{e_1} \longrightarrow \neg f_1$
. $e_1 + e_2 \longrightarrow f_1 \vee f_2$
. $e_1 . e_2 \longrightarrow \exists x, y, z \, (y = x + 1 \wedge z = fin \wedge f_1|_{0,x} \wedge f_2|_{y,z})$.

On a donc bien prouvé ce que l'on voulait,
Donc on a l'équivalence entre RSF et FO.

Nous allons maintenant voir comment passé d'une expression régulière sans $*$ aux automates non-constants.

Exemple :



: cet automate compte modulo 2 et le langage n'est pas sans $*$.

On a l'équivalence entre les langages apériodiques et l'automate non-comptant qui le reconnaît.

Déf : Soit L un langage régulier,
Alors L est apériodique $\Leftrightarrow \exists k \in \mathbb{N}$ tq $\forall u \in \Sigma^*$, u^k est équivalent à u^{k+1}, u^{k+2} ...

On peut aussi dire $\forall u, w$, on a $uv^k w \in L \Leftrightarrow uv^{k+1} w \in L$.

Rmq : L'automate peut compter les v jusqu'à k fois, mais après rien.

Exemple : • $L = \{a^{2n}\}$

Soit $v = a$, $u = \varepsilon$ et $w = \varepsilon$.

On a donc $a \notin L$

$a^2 \in L$

$a^3 \notin L$

$\vdots$

$a^{2n} \in L$

$a^{2n+1} \notin L$

On remarque que ce changement d'appartenance ne se stabilise jamais. On est donc en présence d'un langage périodique.

• $(ab)^*$ est apériodique mais il nous reste à le démontrer.

Théorème : Soit L un langage fini,
Alors L est apériodique.

Démonstration : Soit $k = \max\{|s| \mid s \in L\} + 1$,

Alors si $v \neq \varepsilon$,

Alors $uv^k w$, $uv^{k+1} w \ldots \notin L$

Donc $uv^k w \in L(f) \Leftrightarrow uv^{k+1} w \in L(f)$.

Si $v = \varepsilon$,

Alors $uv^k w = uv^{k+1} w \ldots$

Donc $uv^k w \in L \Leftrightarrow uv^{k+1} w \in L$

Lemme : Soit L un langage apériodique,

Soit $k \in \mathbb{N}$,

Alors $uv^k w \Leftrightarrow uv^{k+1} w \Rightarrow uv^{k+2} w$, et ce quelque soit u, v, w.

Démonstration : $uv^{k+1} w$

Soit $w' = vw$,

On a donc pour u, v, w : $uv^k w' \Leftrightarrow uv^{k+1} w' \in L$,

C'est-à-dire, $uv^{k+1} w \Leftrightarrow uv^{k+2} w$.

CQFD !

Théorème : L, un langage sans $*$,

Alors L est apériodique (Schützenberger).

Démonstration : Nous allons montrer l'implication : SF $\Rightarrow$ apériodique.

On va le montrer par induction structurelle,

On a $\emptyset$, ε, a qui sont finis et donc apériodiques d'après le théorème montré précédemment.

Voyons maintenant pour Σ^*, $(uw) \in \Sigma^*$ est tout le temps vrai,

$(uvw) \in \Sigma^*$ est aussi vrai

Donc $(uv^k w) \Leftrightarrow (uv^{k+1} w)$.

Donc Σ^* est apériodique.

Cas inductifs : Soient L et M, 2 langages apériodiques de csts l et m.

On a donc 3 cas à considérer :

• $\bar{L}$: on prend $k = l$, $uv^k w \in \bar{L} \Leftrightarrow \neg(uv^k w) \in L$,

Or $uv^k w \Leftrightarrow uv^{k+1} w$ par apériodicité de L

Donc $uv^{k+1}w \in I$

CQFD !

- $L \cup M$: on prend $k = \max(l, m)$

$$uv^k w \in L \cup M \iff uv^k w \in L \vee uv^k w \in M$$

$$\Updownarrow$$

$$uv^{k+1} w \in L \cup M \iff uv^{k+1} w \in L \vee uv^{k+1} w \in M$$

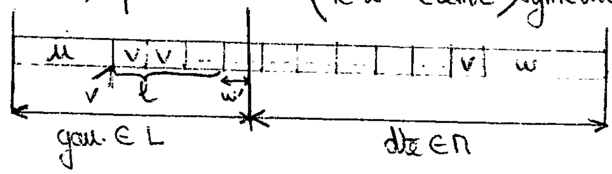
CQFD !

- $L.M$: on prend $k = l + m$,

$uv^k w$: il y a un découpage en 2 sous-mots, soit le mot gauche a $\geq l$ exemplaires, soit le mot droite a $\geq m$ exemplaires de v.

On ne considèrera ici, que le 1er cas (le 2nd étant symétrique).

On a donc



$$\text{gauche} \in L \Rightarrow uv^k w' \in L \Rightarrow uv^{k+1} w' \in L \Rightarrow \text{gauche}' \in L,$$

Donc gauche . droite $\in L.M \Rightarrow uv^{k+1} w \in L.M$.

Il nous faut maintenant montrer que $uv^{k+1} w \in L.M \Rightarrow u.v^k w \in L.M$. Pour cela, on cherche la partie la plus longue, on supprime v dedans.

Donc $uvw' \in L.M$.

CQFD !

Rmq : On ne montrera pas l'implication inverse : L apériodique $\Rightarrow$ L est S.F. C'est beaucoup plus dur.

Corollaire : . On ne peut pas exprimer $\{a^{2n} \mid n \in \mathbb{N}\}$ par une expression sans $*$.

. On ne peut pas exprimer en logique du 1er ordre.

La Théorie des Jeux.

<u>Exemple</u> : Dans un premier temps, la somme est quelconque.

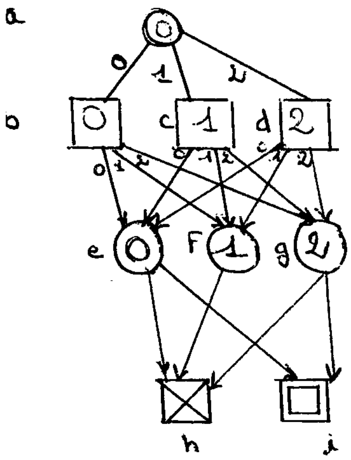
E choisit 0, 1 ou 2

A choisit 0, 1 ou 2

E choisit 0 ou 1

E gagne si la somme finale $\equiv 0 \pmod 3$

On a donc les schémas d'état suivants :



Les 2 autres schémas sont similaires : c'est A qui gagne presque tout le temps.

On peut donc en ressortir une stratégie pour A qui consiste à faire $1 - b_0 - e_1$. Si A se tient à cette stratégie, il gagne à partir des états initiaux quels qu'il soit.

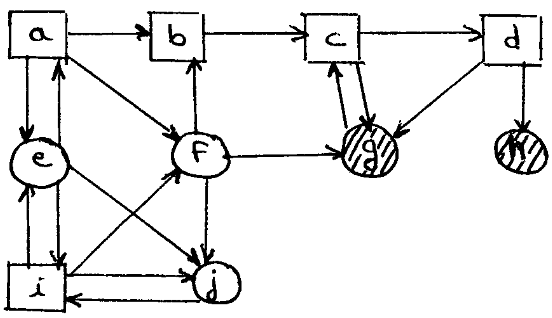
On cherche maintenant les états gagnants pour E : . Init : $\{k\}$ (en 0 coup)

. Iteration 1 : $\{e; g\}$ (en 1 coup)

. Iteration 2 : rien de nouveau.

On a donc, en conclusion, $G_E = \{i; e; g\}$. Tous les autres états sont gagnants pour A.

<u>Exemple</u> :



E veut arriver vers (hachuré), A veut l'éviter. Lorsqu'on est dans □, A choisit un déplacement tandis que le choix appartient à E lorsqu'on est dans ○.

On cherche donc à partir de quels états E peut gagner et comment, ainsi que la même chose pour A. On va donc faire une analyse en arrière.

États gagnants pour E : . Init $\{g, h\}$

. Iteration 1 $\{f, d\}$

. Iteration 2 $\{c\}$

Iteration 3 $\{b\}$

. Iteration 4 : rien de plus

On a donc $G_E = \{b, c, d, f, g, h\}$ et $G_A = \{a, e, i, j\}$

On peut donc exhiber la stratégie gagnante de E : pour gagner de G_E^{K}, E va vers $G_E^{<K}$ (ou Adam est obligé d'aller en $G_E^{<K}$).
La stratégie d'A est plutôt : pour gagner depuis G_A, il faut éviter les états de G_E.

Voyons maintenant un jeux sur un graphe. Prenons un graphe (Q_A, Q_E, Δ) avec $\cdot\, Q_A \cap Q_E = \emptyset$
$\cdot\, Q = Q_A \cup Q_E$
$\cdot\, \Delta \subset Q \times Q$

Exemple :

$Q_A = \{2, 4\}$
$Q_E = \{1, 3\}$
$\Delta = \{12, 21, 23, 34, 43\}$

Note : Un tel graphe est appellé une arène.
Une partie du jeu est un chemin sur l'arène $\in Q^{\omega}$.
La condition de gain est que $G_E \subset Q^{\omega}$. On dit qu'une partie dans G_E est gagnante pour Eve.
Une stratégie pour Eve serait telle que : $s : Q^* \times Q_E \longrightarrow Q$, i.e, si $s(w, q) = q'$, alors $(q, q') \in \Delta$. Autrement dit, pour une histoire wq, Eve fait la transition $s(wq)$.
Exemple : En prenant, pour le graphe précédent, et $G_E = Q_E^* \times F \times Q^{\omega}$
Une stratégie pour Eve serait : $\cdot\, s(w\,1) = 2$.
$\cdot\, s(w\,3) = 4$
Rmq : Cette stratégie est nulle puisque avec $s(w\,3) = F$, Eve aurait gagné tout de suite.
Exemple : En reprenant les mêmes données que ci-dessus, on aurait pu définir la stratégie de Eve avec
$s(w\,1) = 3$
$s(w\,3) = \begin{cases} 4 & \text{si } |w| < 100. \\ F & \text{sinon} \end{cases}$
C'est donc une stratégie avec histoire.
Rmq : On pourrait définir des stratégies sans histoire, i.e, sans w.
Une partie $q_1, q_2, \cdots, q_n$ est conforme à la stratégie s de Eve si $\forall_i \{q_i \in Q_E \mid \qquad \mid i \in Q_E\}$.
Autrement dit, Eve respecte s et Adam fait ce qu'il veut.
Une stratégie pour Eve est gagnante* si chaque partie conforme à cette stratégie** appartient à G_E. (*pour un état initial q_0) (**et commençant à q_0).
Une stratégie s pour Adam est gagnante pour un état initial q_0 si chaque partie conforme à cette stratégie et commençant à q_0 n'appartient pas à G_E.
Un état q_0 est gagnant pour Eve, si Eve a une stratégie gagnante à partir de cet état. On notera $W_E \subset Q$, les états gagnants pour Eve.
Problème : Etant donné une arène (Q_E, Q_A, Δ) et une condition de gain G_E. Trouver W_E, une stratégie pour Eve et W_A une stratégie pour Adam.
Lemme 1 : $W_E \cap W_A = \emptyset$
Démonstration : Soit $q_0 \in W_E \cap W_A$,
Donc $q_0 \in W_E$, d'où Eve a une stratégie gagnante s.

Et $q_0 \in W_A$, d'où Adam a une stratégie gagnante s_A.

On considère la partie P où Eve suit s_E et Adam s_A.

Donc $P \in G_E$ et $P \in G_A$,

Ce qui est contradictoire,

Donc $q W_E \cap W_A \neq \emptyset$.

mme 2 : "Souvent" $W_E \cup W_A = Q$

éfinition : C'est un jeu d'atteignabilité pour Eve et un jeux de sureté pour Adam,

Alors $G_E = Q^* F Q^\omega$ (Eve cherche à visiter F).

héorème : Pour un jeu de type att./sureté $Q^* F Q^\omega$ sur une arène finie :

- $W_E \cup W_A = Q$
- les stratégies sont sans mémoire.
- il y a un algorithme simple pour trouver W_E, W_A, stratégies pour Eve, Adam (cf pred).

Mots infinis, langages et automates.

Un mot infini sur l'alphabet Σ est $a_1, a_2, \ldots, a_n, \ldots$ où $\forall i \in \mathbb{N}$, on a $a_i \in \Sigma$

Un langage* infini sur Σ est un ensemble de mots infinis. (*de mots infinis). On le notera ω_langage.

Exemple : Sur $\Sigma = \{a, b\}$, on a les ω_langages suivants :

1- $\emptyset$

2- $\{abab \ldots ab \ldots\}$ qui contient uniquement le mot $(ab)^\omega$

3- Σ^ω : tous les ω_mots.

4- L_{3} : tous les mots $\#a < \infty$ qui se notera $(b+a)^* b^\omega$.

5- L_4 : après chaque b, il y a quelque part un a qui notera :

Rmq : . Σ^* est un ens. de tous les mots fini (c'est un infini dénombrable $\aleph_0$)

. Σ^ω est un ens. de tous les mots infini (infini non_dénombrable de cardinalité : $2^{\aleph_0}$)

. $\Sigma^* \not\subset \Sigma^\omega$.

Soient les expressions omega régulière définies par, avec $RE = \emptyset \mid a \mid \varepsilon \mid RE + RE \mid RE . RE \mid RE^*$;

$ORE = (RE)^\omega \mid RE . ORE \mid ORE + ORE$.

a sémantique de ORE est donnée par, avec f une RE, g une ORE
- $[\![f]\!] = L \subset \Sigma^*$
- $[\![f^\omega]\!] = \{w_1 w_2 \ldots w_n \ldots\} \forall i \; w_i \in [\![f]\!]\}$.
- $[\![f . g]\!] = \{uv \mid u \in [\![f]\!], v \in [\![g]\!]\}$
- $[\![f + g]\!] = [\![f]\!] \cup [\![g]\!]$ avec f et g, 2 ORE.

. $\subset \Sigma^\omega$ est ω_rationnelle s'il existe une ORE f tq $f = [\![f]\!]$

Nous allons maintenant définir les ω_automates. Ils sont donnés par $A = \langle Q, \Sigma, \Delta, q_0, F \rangle$ (la définition est elle de Büchi).

Un calcul de A sur un mot $w = a_1 a_2 \ldots a_n \ldots$ est représenté par $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots q_n \xrightarrow{a_{n+1}} \ldots$

Un calcul est accepteur s'il visite infiniment souvent F (il n'y a pas de moment à partir duquel on ne passe plus dans)

w est accepté par A s'il existe un calcul accepteur de A sur w.

$L^\omega(A)$ donne tous les ωmots acceptés par l'automate A.

Un langage L est ω_reconnaissable ssi $\exists$ un automate de Büchi A tq $L = L^\omega(A)$

Exemple :

$(a+b)^\omega$

$a^* b b^\omega = a b^\omega$

$(ab)^\omega$

$$= (ab)^\omega + b^\omega + (b^* aab)^\omega + (b^* aa^* b)^* . b^\omega$$

Théorème : Soit L un langage $\subset \Sigma^\omega$,
Alors L est reconnu par un automate de Büchi ssi L peut-être exprimé par une expression ω-régulière.

Démonstration : 1. L est reconnu par un automate de Büchi $\Rightarrow$ L peut-être exprimé par une expression ω-Rég.

Soit ω un mot accepté,
Alors $\exists q_0 \overset{\omega}{\rightsquigarrow} q_i \dots \rightarrow q_n \rightarrow \dots$ un parcours accepteur,
Donc q_0 est initial et F est visité une infinité de fois,
Or F est un ens. fini,
Donc $\exists f \in F$ qui est visité une infinité de fois.
Donc on a $q_0 \rightsquigarrow F \rightsquigarrow F \rightsquigarrow \dots$
Donc on peut écrire ω comme $\omega = u . v_1 v_2 \dots v_n \dots$ tq $q_0 \overset{u}{\rightsquigarrow} f$ et $\forall i \; f \xrightarrow{v_i} f$.
avec $u \in \Sigma^*$ et $v \in \Sigma^+$,
Donc $\omega \in \bigcup_{f \in F} L_{q_0 . F} . (L_{FF} \setminus \varepsilon)^\omega$.
Or $L_{q_0 F}$ est un mot fini régulier, et $(L_{FF} \setminus \varepsilon)$ est un mot régulier finis non-vides,
On décrit $L_{q_0 F}$ et $(L_{FF} \setminus \varepsilon)$ par des RE (finies),
Donc on obtient une ORE pour L.
CQFD.

2. L exprimable par une expression ω-régulière $\Rightarrow$ L reconnu par un automate de Büchi.
On a donc une ORE tq ORE $= (RE)^\omega$ | ORE + ORE | RE . ORE. (RE sans ε).
Il faut donc trouver comment faire un automate de Büchi.

Remarque : Un automate fini est normalisé s'il a un seul état initial, un seul état final, que rien ne entre dans l'état initial et que rien ne sort de l'état final.

Lemme : Tous les automates finis quelconques sont équivalents à des automates normalisés.

Nous avons 3 cas à traiter.

Cas 1 : g^ω où g est une RE,
Pour g il y a donc un automate, et donc, d'après le lemme précédent un automate normalisé.
On a donc un automate g^ω qui reboucle de f vers tous les successeurs de l'état initial.

Cas 2 : Soient h et g 2 ORE,
Soient A et B les 2 automates de Büchi pour leur ω-langages.

Remarque : Un automate de Büchi n'a qu'un seul état initial isolé. On montrera donc qu'une ORE peut être reconnu par un automate de Büchi normalisé.

Donc notre automate pour $(h+g)$ sera de la forme :

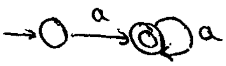
Cas 3 : Soit h un RE et g une ORE,
Soient A et B leurs automates,
On cherche donc un automate pour A.B qui sera de la forme :

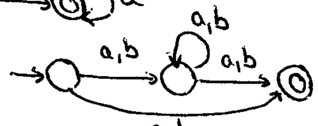
CQFD

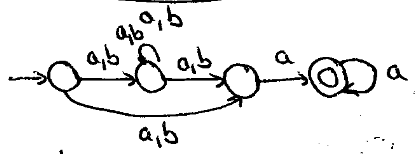
On a bien démontré l'équivalence.

Exemple : Construisons l'automate pour $(a+b)^* . a^\omega$

a : 

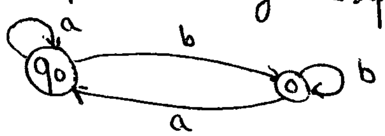
a^ω 

$(a+b)^+$: 

$(a+b)^+ a^\omega$ 

Remarque : On obtient un automate lourd à construire mais qui a l'avantage d'être simple à construire.

Exemple : Déterminons l'expression ω_régulière qui est reconnu par l'automate suivant :



$L = L_{q_0F} . (L_{FF})^\omega$

$L = a^* b (b^+ a^+ b^+)^* . (b^+ (a+b^+)^+ + b^+)^\omega$

Là encore, on obtient une expression compliquée. Une expression équivalente plus simple serait donc :

$L = L_{q_0F (\text{sans repasser par } F)} . (L_{FF (\bar{m}\ \text{cond.}) \setminus \varepsilon})^\omega$.

D'où $L = a^* b\ (b + a^+ b)^\omega$.

Terminologie : Un langage ω_reconnaissable par un automate de Büchi,
⟺ un langage ω_rationnel est exprimable par une ORE,
⟺ un langage ω_régulier.

Théorème : Les langages réguliers sont clos par $\cap, \cup, -, f, f^{-1}$.

Commentaires : . Si L et M sont ω_langages de A et B,
on sait faire un automate pour $L \cup M$.
. Pour les autres, pareil

Remarque : . On ne peut pas déterminiser les automates de Büchi. Les déterminisés sont moins expressifs que les déterminisés.
. Le complément d'un langage ω_régulier est ω_régulier. C'est un théorème de Büchi mais la preuve est difficile.

Théorème : L est un langage ω_régulier ssi L peut-être défini en logique MSO.

Démonstration : La même que pour les mots finis.

Exemple : • $\#b < \infty$ ou $(a+b)^* a^\omega$ s'écrit en logique MSO :

$$\exists x \forall y \, (y > x, \; P_a(y)).$$

• $(b(aa)^*b)^\omega$ qui donne en MSO :

$$\exists P, Q, R,$$

$P(0)$ ← initialisations

$\forall x \quad P(x) \wedge b(x) \Rightarrow Q(S(x))$

$\quad Q(x) \wedge a(x) \Rightarrow R(S(x))$

$\quad \vdots$

$\quad$ pour toutes les transitions.

$\Big\}$ transitions

$\forall z \, \exists y (y > z) \wedge P(y)$ ← conditions de Büchi.