Automates finis et Expressions régulières

Nadji Gauthier `Nadji.Gauthier@inria.fr`

October 13, 2004

Références théoriques et explications : `http://www.pps.jussieu.fr/~rifflet/IF241/` (cours 1 et 2).

1 Rappel

Un *alphabet* A est un ensemble. Une *lettre* est un élément d'un alphabet. Un *mot* est une suite de lettres. "m est un mot sur A" s'écrit formellement $m \in A^*$. Un *language* est un ensemble de mots. Le mot vide se note ϵ. On fera l'abus de notation qui consiste à considerer une lettre comme un mot de longueur 1.

Par défaut dans la suite on considère que l'aplhabet A est a, b.

Si m et m' sont deux mots, on note $m.m'$ leur concaténation. Exemples : $a.aab = aaab$, $\epsilon.m = m$, $m.\epsilon = m$ On note m^n le mot qui est la concaténation de n mots m :

$$\begin{array}{lcl} m^0 & = & \epsilon \\ m^{(n+1)} & = & m^n.m \end{array}$$

On note m^* le language ou les mots sont des concaténations de m : $m^* = \{m^n / n \in \mathbb{N}\}$. On note $|m|_x$ où x est une lettre le *nombre d'occurence* de x dans m:

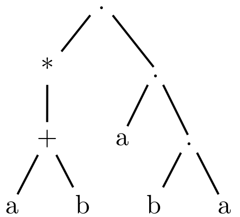
$$\begin{array}{lcll} |\epsilon|_x & = & 0 & \\ |m.x|_x & = & |m|_x + 1 & \\ |m.y|_x & = & |m|_x & \text{si } x \neq y \end{array}$$

On note r une *expression régulière* sur un alphabet A, dont la syntaxe est :

$$\begin{array}{lcll} r & := & \epsilon & \\ & | & a & a \in A \\ & | & r.r & \\ & | & r + r & \\ & | & r^* & \end{array}$$

Cette notation signifie qu'une expression régulière est un arbre, dont les feuilles sont étiquettées par des lettres de A ou par ϵ, et dont les noeuds sont étiquettés par .,+ ou $*$. Si un noeud est étiquetté par . ou par $+$, alors il possède deux fils (l'ordre est important). Si un noeud est étiquetté par *, alors il a un seul fils. On écrira cependant les expressions régulière avec la notation linéaire habituelle(comme pour les expressions arithmétiques).

Exemple : l'expression régulière suivante



s'écrit plus naturellement $(a+b)^*aba$, avec la convention que la concaténation (.) ne s'écrit pas, la concaténation a priorité sur la disjonction, et l'étoile la plus forte priorité (comme le produit, la somme et la puissance en mathématiques).

Une expression régulière dénote un langage que l'on peut définir récursivement par rapport à sa structure. Soit $\llbracket r \rrbracket$ le langage dénoté par r sur l'alphabet A. On le définit comme suit :

$$\begin{array}{lcl}
\llbracket \epsilon \rrbracket & = & \{\epsilon\} \\
\llbracket a \rrbracket & = & \{a\} \\
\llbracket r_1 + r_2 \rrbracket & = & \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket \\
\llbracket r_1 . r_2 \rrbracket & = & \{m.m' / m \in \llbracket r_1 \rrbracket, m' \in \llbracket r_2 \rrbracket\} \\
\llbracket r^* \rrbracket & = & \{m^n / m \in \llbracket r \rrbracket, n \in \mathbb{N}\}
\end{array}$$

2 Automates finis non déterministes

2.1 définitions

Un *automate fini non déterministe* (NFA, *non deterministic finite state automaton*) sur un alphabet A est un triplet $< i, f, T >$ où $i \in \mathbb{N}, f \in \mathbb{N}, T \subset \mathbb{N} \times (A \cup \epsilon) \times \mathbb{N}$. i et f sont des *états* (souvent plus formellement on utilise un ensemble Q abstrait au lieu de $\mathbb{N}$). T est un ensemble fini de *transitions.* Soit t une transition de T. Si $t = (s, a, s')$, on appelle a l'*étiquette* de t. Sinon, si $t = (s, \epsilon, s')$ on appelle t une *ϵ-transition* .

Un automate peut se décrire à l'aide d'une matrice de transition telle que les lignes correspondent à chacun de ses états, que les colonnes correspondent à chaque lettre de l'alphabet et l'intersection d'une ligne représentant l'état s avec une colonne représentant x (x étant une lettre a ou ϵ) il y a l'ensemble des états s' tel que l'automate possède une transition (s, x, s'). L'état initial est marqué par une flèche et le final par une étoile.

Exemple :

	ϵ	a	b
$\rightarrow 1$	2	2	
2		3	2
$(*)3$			

code l'automate $< 1, 3, \{(1, a, 2), (1, \epsilon, 2), (2, b, 2), (2, a, 3)\} >$

Un *chemin* d'un état s vers s' dans un NFA est une suite de transitions dans T de la forme $(s_0, x_0, s_1), (s_1, x_1, s_2), \dots (s_{n-1}, x_{n-1}, s_n)$. On appelle alors n la *longueur du chemin*, et le mot $x_0.x_1 \dots .x_{n-1}$ son *étiquette.*

Le *langage reconnu par un NFA* est l'ensemble des étiquettes des chemins partants de l'état initial vers l'état final.

Exemple : l'automate précédent reconnait le langage décrit par l'expression régulière $(a|\epsilon)b^*a$. L'étiquette de $(1,\epsilon,2)(2,b,2)(2,b,2)(2,a,3)$ est bba

2.2 construction de Thompson

C'est un algorithme qui permet de construire un NFA reconnaissant le langage décrit par une expression régulière. On peut le voir comme une fonction récursive *thompson* prenant en paramètre une ER et retournant un NFA. Il est facile de constater que le langage reconnu par un automate ne dépend pas des entiers choisis pour numéroter les états mais uniquement de leurs relations. C'est pour cela que l'on peut toujours choisir un ensemble d'entiers dans lequel aucun des états d'un certain automate n'apparait. Donc étant donné deux automates N_1 et N_2 , on peut toujours choisir une numérotation pour que N_2 n'ait aucun état en commun avec N_1. On note le fait que deux automates N_1 et N_2 ne possèdent aucun état en commun par $N_1\#N_2$ (se lit N_1 disjoint de N_2). On note le fait qu'un automate A n'aie pas d'état dans un ensemble E par $A\#E$. On écrit $N_1\#N_2\#E$ pour $N_1\#N_2$ et $N_1\#E$ et $N_2\#E$.

$$\begin{array}{lcl}
\textit{thompson}\ (\epsilon) & = & <1,2,\{1,\epsilon,2\} \\
\textit{thompson}\ (a) & = & <1,2,\{(1,a,2)\}> \\
\textit{thompson}\ (r_1+r_2) & = & <i,f,T_1\cup T_2\cup\{(i,\epsilon,i_1),(i,\epsilon,i_2),(f_1,\epsilon,f),(f_2,\epsilon,f)\} \\
 & & \text{avec } N_k=<i_k,f_k,T_k>=\textit{thompson}\ (r_k), N_1\#N_2\#\{i\}\#\{f\} \\
\textit{thompson}\ (r_1.r_2) & = & <i_1,f_2,T_1\cup T_2\cup\{(f_1,\epsilon,i_2)\} \\
 & & \text{avec } N_k=<i_k,f_k,T_k>=\textit{thompson}\ (r_k), N_1\#N_2 \\
\textit{thompson}\ (r^*) & = & <i',f',T\cup\{(i',\epsilon,i),(i',\epsilon,f'),(f,\epsilon,i),(f,\epsilon,f')\} \\
 & & \text{avec } N=<i,f,T>=\textit{thompson}\ (r_k), N\#\{i'\}\#\{f'\}
\end{array}$$

2.3 Décider si un mot est dans le langage d'un NFA

Etant donnés un mot m et un NFA $N=<i,f,T>$, le problème est donc de savoir si il existe un chemin de i vers f tel que son étiquette soit m. C'est une variante de l'algorithme d'accessibilité dans un graphe.

Supposons qu'il existe un tel chemin.

Si m est ϵ, alors le chemin est soit vide et on a $i=f$, soit de la forme $(i,\epsilon,s_1),(s_1,\epsilon,s_2),\ldots,(s,\epsilon,f)$. Pour répondre au problème, il suffit alors de regarder tous les états vers lesquels on peut aller à partir de i en suivant uniquement des ϵ-transitions.

Si m s'écrit $a.m'$, alors le chemin est de la forme $(i,a,s),c$ ou $(i,\epsilon,s_1),(s_1,\epsilon,s_2),\ldots,(s_n,a,s),c$ où c est un chemin de s vers f dont l'étiquette est m'.

On peut donc généraliser un peu le problème et le résoudre de manière inductive sur la longueur de m. On va chercher à décider si étant donné un ensemble d'états S, il existe un chemin partant d'un état de S et se terminant en f telle que son étiquette soit m. On écrit ce problème $C(S,m)$. On se ramène ainsi au problème de départ en considérant $C(\{i\},m)$.

Si m est ϵ, alors la question est de savoir si $f \in S$ ou s'il existe un chemin de la forme $(s_0, \epsilon, s_1), (s_1, \epsilon, s_2), \ldots, (s_n, \epsilon, f)$ tel que $s_0 \in S$. Soit S' l'ensemble contenant S et tous les états vers lesquels on peut aller à partir d'un état de S en suivant uniquement des ϵ-transitions. On appelle S' la *clôture par ϵ-transition* de S. Les états de S' sont dit ϵ-accessibles à partir de S. Si on a calculé S', $C(S, \epsilon)$ est équivalent à $f \in S'$.

Si m est am', alors un chemin de S vers f s'écrit forcément $(s, a, s'), c$ ou $(s, \epsilon, s_1), (s_1, \epsilon, s_2), \ldots, (s_n, a, s'), c$ avec $s \in S$ et c un chemin vers f dont l'étiquette est m'. Autrement dit, ce chemin se termine par une transition étiquettée par a dont l'origine est dans la *clôture par ϵ-transition* de S (que l'on note de nouveau S'). Donc c commence par un état (s') qui est la destination d'une transition étiquettée par a et dont l'origine est dans S'. Plus formellement, c commence par un état dans $S_a = \{s'/(s, a, s') \in T, s \in S'\}$. Remarquons que si S_a est vide, alors il ne peut pas y avoir de chemin de S vers f étiquetté par m puisqu'il devrait contenir un état dans S_a. Donc il existe un chemin de S vers f étiquetté par m si et seulement si il existe un chemin de S_a vers f étiquetté par m'. C'est à dire qu'on se ramène à un problème plus petit : $C(S, m) \Leftrightarrow C(S_a, m')$.

La définition de la clôture par ϵ-transition est :

$$\begin{array}{l} \mathit{epsclosure}_N\ (S) = \left\{ \begin{array}{ll} S & \text{si } S' \subseteq S \\ \mathit{epsclosure}_N\ (S \cup S') & \text{sinon} \end{array} \right. \\ \text{avec } S' = \{s'/s \in S, (s, \epsilon, s') \in T\} \end{array}$$

Remarquons que cette fonction récursive est bien calculable sur tout ensemble fini d'états puisqu'il n'y a qu'un nombre fini de transitions (donc d'états dans l'automate). A chaque étape on peut calculer S' car T est fini, on peut tester si $S \subseteq S'$ car S et S' sont finis et on se rappelle sur un nombre d'états strictement plus grand.

Pour l'algorithme de reconnaissance, on a besoin d'une fonction auxilliaire *multitrans* :

$$\mathit{multitrans}_N(a, S) = \{s'/s \in S, (s, a, s') \in T\}$$

La définition de l'algorithme de reconnaissance est donc :

$$\begin{array}{lcl} \mathit{reco}_{nfa}\ (S, \epsilon) & = & \mathit{true}, \text{si } f \in \mathit{epsclosure}\ (S) \\ \mathit{reco}_{nfa}\ (S, \epsilon) & = & \mathit{false}, \text{si } f \notin \mathit{epsclosure}\ (S) \\ \mathit{reco}_{nfa}\ (S, a.m) & = & \mathit{reco}_{nfa}\ (\mathit{multitrans}(a, \mathit{epsclosure}\ (S)), m) \end{array}$$

3 Automates finis déterministes

Un *automate fini déterministe* (DFA, *deterministic finite state automaton*) sur un alphabet A est un triplet $< i, F, T >$ où $i \in \mathbb{N}, F \subset \mathbb{N}, T \subset \mathbb{N} \times A \times \mathbb{N}$. De plus, T doit être *déterministe*, c'est à dire

$$\forall s, a, s_1, s_2.\ \{(s, a, s_1), (s, a, s_2)\} \subseteq T \Rightarrow s_1 = s_2$$

. Autrement dit, pour un état s et une lettre a donnée il ne peut y avoir qu'une seule transition partant de s étiquettée par a. Si pour tout état de l'automate et toute lettre de A il y en a toujours une, on dit que l'automate est *complet*. Notons que pour un DFA il peut y avoir plusieurs états finaux. La définition de chemin et d'étiquette de chemin est identique à celle d'un NFA. Le langage reconnu par un DFA est l'ensemble des étiquettes des chemins partant de l'état initial est arrivant dans un des états finaux.

L'algorithme de reconnaissance devient trivial du fait du déterminisme. Un des intérêts des DFA est qu'ils peuvent représenter les même langages que les NFA (il est facile de se convaincre de la propriété symétrique). On va le montrer constructivement.

Soit une fonction injective *code*: $\mathcal{P}(\mathbb{N}) \rightarrow \mathbb{N}$ (rappel : $code\ (S) = code\ (S') \Rightarrow S = S'$). Soit un certain NFA $N =< i, f, T >$. Le DFA suivant :

$$\left\langle \begin{array}{l} code\ (epsclosure_N\ (\{i\})), \\ \{code\ (S)/f \in S\}, \\ \{code\ (S), a, code\ (multitrans_N(a, epsclosure_N\ (S)))\} \end{array} \right\rangle$$

reconnait le même langage que N. L'intuition est que chaque étape de reconnaissance dans le DFA correspond à une étape de reconnaissance dans le NFA avec la fonction $reco_{nfa}$.

En pratique on ne calcule qu'une partie de ce DFA car il y a beaucoup d'états qui ne servent à rien dans la description mathématique. Une façon de procéder est décrite par la fonction *det*. Elle part d'un NFA N, un ensemble d'état S qui peut être présenté à $reco_{nfa}$, une fonction *code* donnant un identifiant unique à un ensemble d'état, et un ensemble de transitions E dans le nouveau DFA. Elle ajoute à *code* tous les ensembles d'états sur lesquels on peut accéder dans $reco_{nfa}$ qui ne sont pas déjà dans le domaine de *code*. Un invariant est que toutes les transitions à partir d'un ensemble d'état présent dans le domaine de *code* sont déjà dans E ou vont y être ajoutés. Elle ajoute à E, si nécessaire, toutes les transitions simulant une étape de $reco_{nfa}$ à partir de la clôture de S. Elle renvoie le *code* correspondant à la clôture de S.

$$\begin{array}{lr} det\ (N, S, code, E) = code, E, code\ (S') & \text{si } S' \in dom\ (code) \\ & \text{avec } S' = epsclosure\ (S) \end{array}$$

$$\begin{array}{lr} det\ (N, S, code, E) = code_k, T_k \cup \{(id, a_n, s_n)/1 \leq n < k\}, id & \text{sinon} \\ & \text{avec } S' = epsclosure\ (S) \\ & id \notin img\ (code) \\ & \{(a, S_a)/S_a = multitrans_N(a, S')\} = \{(a_1, S_1), \ldots (a_k, S_k)\} \\ & (code_0, T_0) = (code \cup \{(S', id)\}, E) \\ & (code_{n+1}, T_{n+1}, s_{n+1}) = det\ (N, S_{n+1}, code_n, T_n) \end{array}$$

3.1 Minimisation d'un DFA

Les DFA que l'on obtient après la déterminisation d'une construction de Thompson d'une ER ont souvent des états redondants, c'est à dire des états à partir desquels on peut reconnaitre le même langage. En pratique on cherche à les éliminer (en fait les fusionner entre eux).

Exemple : Dans le DFA suivant:

	a	b
$\rightarrow 1$	2	3
2	2	3
$(*)3$		

l'état 1 est redondant avec l'état 2, on pourrait les fusionner pour obtenir ce DFA reconnaissant le même langage :

	a	b
$\rightarrow 1$	1	2
$(*)2$		

Une technique est de chercher à distinguer progressivement les états. On va chercher à calculer les couples d'états tel qu'il existe un chemin de longueur n qui partant de l'un des deux états arrive dans un état final et partant de l'autre n'y arrive pas, ou qu'un chemin existe depuis l'un et pas depuis l'autre.

Soit $D =< i, F, T >$ un DFA ayant k états. On appelle classiquement δ la *fonction de transition* (partielle) de D :

$$\delta(s, a) = s' \text{ avec } (s, a, s') \in T$$

On étend δ sur les mots :

$$\begin{array}{lcl} \delta(s, \epsilon) & = & s \\ \delta(s, a.m) & = & \delta(\delta(s, a), m) \end{array}$$

Si il existe un mot m de longueur n tel que $\delta(s, m)$ est défini et $\delta(s', m)$ n'est pas défini ou que $\delta(s, m)$ et $\delta(s', m)$ sont définis mais $\delta(s, m) \in F$ et $\delta(s, m) \notin F$ on dit que *m sépare s et s'* et que s et s' sont *n-distinguable.*

Pour déterminer si deux états reconnaissent le même language, on va calculer récursivement tous les états n-distinguables pour n de 0 à k.

Remarque : si deux états ne sont pas k-distinguables, alors pour tout k', ils ne sont pas k'-distinguables.

On défini récursivement les états n-distinguables:

$$\begin{array}{lcl} M_0 & = & \{(s, s')/s \in F, s' \in F'\} \\ M_{n+1} & = & M_n \cup \{(s, s')/\exists a.(\delta(s, a) \text{ est défini et } \delta(s', a) \text{ non défini })ou(\delta(s, a), \delta(s', a)) \in M_n\} \end{array}$$

En pratique on ne calcule que la moitié de ces ensembles car ils sont symétriques. L'ensemble $M = M_k$ contient tous les états distinguables. On peut alors pour chaque état associer son *représentant* qui reconnait le même langage :

$$repr(s) = min_{s'} \ \{(s, s') \notin M\}$$

Un DFA équivalent à D est alors :

$$< repr(i), \{repr(s)/s \in F\}, \{(s, a, repr(s'))/s = repr(s), (s, a, s') \in T >$$

On peut facilement étendre cette technique pour décider si deux automates reconnaissent le même langage.

3.2 Calcul d'une expression régulière décrivant le langage reconnu par un automate

3.2.1 Méthode de McNaughton et Yamada

Soit le DFA $D =< s, F, T >$.

$$\begin{array}{ll} L_{ij0} = a_1 + \ldots + a_n & \{a_1, \ldots a_n\} = \{a/(i, a, j) \in T\} \\ L_{ijk} = L_{ij(k-1)} + L_{i(k-1)(k-1)}.L^*_{(k-1)(k-1)(k-1)}.L_{(k-1)j(k-1)} k > 0 & \end{array}$$

L_{ijk} est l'ensemble des étiquettes des chemins de i à j qui passent par des états strictement inférieurs à k. Si $k = 0$, cela veut dire que le chemin ne passe par aucun état, donc le langage est décrit simplement par l'union de toutes les lettres qui étiquettent les transitions de i à j. Si $k > 0$, alors un chemin de i à j passant par des états strictment inférieurs à k est soit un chemin qui passe par des états uniquement inférieurs à $k - 1$, soit il peut se découper en un chemin qui va vers $k - 1$, fait quelques boucles en passant par par des états strictement inférieurs à $k - 1$, puis va de $k - 1$ à j.

Le langage reconnu par D est alors :

$$L(D) = \bigcup_{f \in F} L_{sfn}$$

où n est l'état maximal dans D. Dans cet algorithme, les langages sont soit vides, soit décrits par des ER. On étend facilement les opération d'union, concaténation et d'étoile à ces objets.

3.2.2 Méthode d'effacement d'états

On définit un *automate étendu* (AE) par le triplet $< i, f, T >$ où $i \in \mathbb{N}, f \in \mathbb{N}$ et $T \subseteq \mathbb{N} \times r \times \mathbb{N}$. Autrement dit, les étiquettes des transitions sont des ER. On définit le langage reconnu par un automate étendu comme l'union des langages décrits par la concaténation des étiquettes des chemins de i à f. Remarquez qu'un NFA est un cas particulier d'automate étendu.

Etand donné un automate étendu $D =< i, f, T >$ on va construire un automate étendu D' qui va reconnaitre le même langage, mais avec un état de moins. Soit e un état de D qui n'est ni final ni initial, avec une boucle sur lui même étiquettée par r_e. Un chemin de i à f qui passe par e passe alors forcément par un prédécesseur de e et par un successeur de e. Donc le langage reconnu par ce passage est la concaténation des langages reconnus par un des prédécesseur, r_e^*, et un des successeurs. On exploite ce fait pour éliminer e.

Les langages des prédécesseurs sont décrits par

$$P = \{(s, r)/(s, r, e) \in T\} = \{(s_1, r_1), \ldots, (s_k, r_k)\}$$

et ceux des successeurs sont décrits par

$$S = \{(r', s')/(e, r, s) \in T\} = \{(r'_1, s'_1), \ldots, (s'_{k'}, r'_{k'})\}$$

Un AE qui reconnait le même langage que D est alors :

$D' =< i, f, T' = \{(s, r, s')/s \neq e$ et $s' \neq e\} \cup \{(s_a, r_a.r_e^*.r'_b, s'_b)/1 \leq a \leq k$ et $1 \leq b \leq k'\}$

On peut de plus fusionner les transitions (*simplification*) pour avoir une représentation plus compacte:

$$D'' =< i, f, \{(s, r_u, s')/r_u = \cup_r \{(s, r, s') \in T'\}$$

Si on itere ce processus, on tombe dans l'un des deux cas suivants :

- il ne reste qu'un seul état, $i = f$, et alors après simplification on a $T = \{(i, r, f)\}$ donc le langage reconnu par cet automate est r^*
- il ne reste que deux états, et alors après simplification on a $T = \{(i, r_i, i), (i, r_{if}, f), (f, r_f, f), (f, r_{fi}, f)\}$ et on peut vérifier que le langage reconnu par cet automate est

$$r_i^* r_{if} r_f^* (r_{fi} r_i^* r_{if} r_f^*)^*$$

4 Exercices

1. Soit $A = a, b$. Donner l'expression régulière qui correspond aux langages suivants :
   (a) $\{m \in A^*/$m se termine par aba$\}$
   (b) $\{m \in A^*/$m contient aba ou m contient baa$\}$
   (c) $\{m \in A^*/|m|_a$ est pair$\}$
   (d) $\{m \in A^*/$m ne se termine pas par ba$\}$
   (e) $\{m \in A^*/$m ne contient par ba$\}$
   (f) $\{m \in A^*/|m|_a > 0$ et $|m|_b > 0$ $\}$
   (g) $\{m \in A^*/m$ contient au plus une fois le mot $aa\}$
   (h) $\{m \in A^*/$tout sous mot aa dans m est immédiatement précédé par $bb\}$
2. (*) Donner un DFA qui reconnait les langages suivants:
   (a) $\{m \in A^*/$chaque block de 5 lettres consécutives de m contienne au moins deux a$\}$
   (b) $\{m \in A^*/$la 4ème lettre en partant de la fin de m soit un a$\}$
   (c) $\{m \in A^*/|m|_a$ soit divisible par 4 et $|m|_b$ soit divisible par 3$\}$
3. (*) Soit le DFA suivant (la flèche désigne l'état initial et l'étoile un état final):

	a	b
$\rightarrow 1$	1	2
$(*)2$	2	1

   Décrire le langage reconnu par une expression régulière.

4. (*) Soit le DFA suivant:

	a	b
$\rightarrow (*)1$	2	1
$(*)2$	3	1
3	3	3

Idem.

5. Convertir ce NFA en DFA:

	a	b
$\rightarrow 1$	$2, 4$	2
$(*)2$	3	$2, 3$
3	4	1
$(*)4$		1

6. (*) Convertir ce NFA en DFA:

	a	b
$\rightarrow 1$	$1, 2$	1
2	3	3
3	4	
$(*)4$	4	4

7. (*)Donner un NFA qui reconnait ces langages, sur $A = \{0, 1, 2, 3, 4, 5\}$:
   (a) $\{m \in A^*/$le dernier chiffre de m est déjà apparu avant la dernière position$\}$
   (b) $\{m \in A^*/$le dernier chiffre de m n'est jamais apparu avant$\}$
   (c) $\{m \in A^*/$il existe deux 0 dans m qui sont séparés par un nombre de lettres multiple de 5$\}$
   (d) $\{m \in A^*/$si m est interprété comme un entier, $m > 31\}$
   (e) $\{m \in A^*/$si m est interprété comme un entier, $12 < m < 40\}$

8. soit le NFA suivant:

	ϵ	a	b	c
$\rightarrow 1$	$2, 3$		2	3
2		1	3	$1, 2$
$(*)3$				

   (a) pour chaque état i, donner les états ϵ-accessible à partir de i
   (b) donner tous les mots de longueur inférieure à 3 acceptés par l'automate
   (c) convertir cet automate en DFA

9. (*) Faire la même chose avec le NFA suivant:

	ϵ	a	b	c
$\rightarrow 1$		1	2	3
2	1	2	3	
$(*)3$	2	3		1

10. Soit le DFA suivant:

	a	b
$\rightarrow 1$	2	1
2	3	1
$(*)3$	3	2

    (a) donner les expressions L_{ij0}, puis les expressions L_{ij1} et L_{ij2} en les simplifiants au maximum
    (b) donner une expression régulière pour le langage reconnu par ce DFA

11. Faire pareil avec celui là :

	a	b
$\rightarrow 1$	2	3
2	1	3
$(*)3$	2	1

12. Donner une expression régulière grâce à la méthode d'élimination des états pour les deux automates précédents

13. Et pour celui ci aussi :

	a	b
$\rightarrow 1$	4	1
2	1	4
3	3	2
4	2	3

14. Convertir ces expressions régulières en nfa avec la méthode de Thompson :
    (a) $a(a+b)^*$
    (b) $(a+ba^*b)^*$

    Donner un DFA qui les reconnait.

15. minimiser l'automate suivant:

	a	b
$\rightarrow 1$	2	1
2	1	3
3	4	2
$(*)4$	4	1
5	4	6
6	7	5
7	6	7
8	7	4

5 Programmation

On peut considérer une expression régulière comme suit :

```
type re = | Eps
  | Ltr of char
  | Conc of re * re
  | Star of re
  | Union of re * re
}
```

1. Programmer une fonction

```
pretty_print : re -> string
```

telle que

```
pretty_print r
```

retourne une représentation textuelle de r. Attention à la précédence des opérateurs, il faut introduire des parenthèses. Si possible, faire cette fonction telle que le nombre de parenthèses soit minimal.

2. Programmer les fonctions suivantes :

```
val contains_eps : re -> bool
val only_eps : re -> bool
val is_language_finite : re -> bool
```

```
contains_eps r
```

doit décider si r dénote un langage contenant le mot vide.

```
only_eps r
```

doit décider si r dénote *le* langage contenant uniquement le mot vide.

```
is_language_finite r
```

doit décider si r dénote un langage fini

Quelle est la complexité de chacune de ces fonctions ?

3. Définir un type nfa pouvant représenter un NFA résultant de la construction de Thompson. On pourra prendre par exemple :

```
type nfa = { start : int
   ; stop : int
   ; trans : (int * (char option) * int) list
   }
```

ou celui ci :

```
type trans_nfa = TEps of int list | TLtr of char * int

type nfa = { start : int
   ; stop : int
   ; trans : (int * trans_nfa) list
   }
```

Puis coder la construction de Thompson :

```
val thompson : re -> nfa
```

Quels sont les différences entre les deux versions de nfa ? Quelle est la complexité de votre fonction ? Coder une fonction thompson de complexité optimale. S'arranger pour que l'accès aux transitions d'un état soit en $O(1)$.

Pour ceux qui ne savent pas comment démarrer, codez une fonction

```
thompson_rec : int -> re -> nfa
```

telle que

```
thompson_rec n r
```

retourne un nfa dont l'état de départ est n et qui ne crée pas d'états inférieurs à n.

4. Coder une fonction

```
possible_trans : nfa -> int list -> int list * (char * int) list
```

telle que

```
 possible_trans nfa stlist
```

retourne une paire (stlist', transposs) telle que stlist' soit la liste de tous les états ϵ-accessibles à partir de stlist et transposs soit toutes les transitions étiquettées dont l'origine est dans stlist'. Quelle est la complexité de cette fonction ? Comment faire une fonction de complexité optimale ?

Pour ceux qui ne savent pas comment démarrer, codez une fonction

```
add_trans : int list -> (char * int) list -> int list ->
            int list * (char * int) list
```

telle que

```
add_trans seen trans slist
```

ajoute à seen les états ϵ-accessibles à partir de slist qui n'y sont pas déjà et ajoute à trans toutes les transitions étiquettées dont l'origine est dans un état ϵ-accessible à partir de slist, mais pas dans seen .

5. Coder une fonction

```
reco_nfa : nfa ->string -> bool
```

telle que

```
reco_nfa nfa s
```

décide si s est dans le langage reconnu par nfa.

Quelle est la complexité de cette fonction ? Comment faire une fonction de complexité optimale (non trivial) ?

6. Un DFA peut se représenter ainsi :

```
type dfa = { final : bool array
   ; trans : ((char, int) Hashtbl.t) array
   }
```

Faire une fonction

```
val determinize : nfa -> dfa
```

qui déterminise le nfa. Quelle est la complexité de votre fonction ? (difficile)Pouvez vous en donner une version en $O(s * t * n \log n)$ où s est le nombre d'états dans le DFA et n le nombre d'états dans le NFA et t le nombre maximal de transition à partir d'un état ?

7. Coder une fonction qui prend un dfa en argument et retourne une expression régulière dénotant le langage reconnu par ce dfa

8. (optionel)Coder une fonction qui prend un dfa en argument et retourne un dfa minimal reconnaissant le même langage

9. (optionel)Coder une fonction qui décide si deux dfa sont équivalents (on pourra la tester sur des automates et leur minimisés)

10. Coder une fonction qui prend deux DFA et retourne un DFA qui reconnait les mots qui sont dans l'intersection des langages reconnus par les deux DFA en arguments. Faire la même avec deux NFA.