Option Informatique en Spé MP et MP*
DS du 27 octobre 1999 : le corrigé

Problème 1

Question 1 • Soient L un langage reconnaissable, et $\mathcal{A} = (Q, \delta, i, F)$ un automate fini reconnaissant L ; notons $N = |Q|$. Considérons un mot u vérifiant $|u| \geqslant N$. Au cours de la lecture de ce mot, l'automate passe par $N+1$ états au moins. D'après le principe des tiroirs, il existe au moins un état j par lequel l'automate passe au moins deux fois. Ceci revient à dire que u peut se décomposer en xyz, avec $y \neq \varepsilon$ et $\delta^*(i, xy) = \delta^*(i, x) = j$. Une récurrence immédiate montre que $\delta^*(i, xy^n) = j$; on en déduit $\delta^*(xy^nz) = \delta^*(xyz)$ et donc $xy^nz \in L$ pour tout $n \in \mathbb{N}$.

• La condition $|xy| \leqslant N$ sera obtenue en choisissant j comme suit : pour $0 \leqslant k \leqslant |u|$, notons $q_k = \delta^*(i, u_1u_2 \dots u_k)$ l'état dans lequel se trouve l'automate après lecture du préfixe de longueur k de u ; alors $j = \min\{k : 0 \leqslant k \leqslant |u| \text{ et } \operatorname{Card}\{q_0, q_1, \dots, q_k\} \leqslant k\}$. Autrement dit : tant que l'on n'a pas lu j lettres de u, on n'est pas passé deux fois par le même état.

Question 2 • Procédons par l'absurde. Supposons L_1 reconnaissable : le lemme de l'étoile affirme l'existence d'un naturel N tel que tout mot $u \in L_1$, de longueur au moins égale à N, se décompose en $u = xyz$ avec $y \neq \varepsilon$, $|xy| \leqslant N$ et $xy^nz \in L_1$ pour tout $n \in \mathbb{N}$. Comme $(N+2)! \geqslant N$, nous pouvons appliquer ce lemme au mot $u = a^{(N+1)!}$; soit donc xyz la factorisation de u dont le lemme affirme l'existence ; on aura $1 \leqslant y \leqslant N$, donc $(N+1)! + 1 \leqslant |xy^2z| \leqslant (N+1)! + N$. Il est clair que $(N+1)! < (N+1)! + 1$; par ailleurs, $N+1 > N$ et $(N+1)! \geqslant 1$ impliquent :

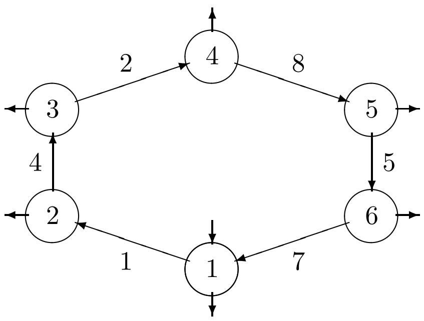
$$(N+2)! = (N+2)(N+1)! = (N+1)! + (N+1)(N+1)! > (N+1)!$$

On a donc $(N+1)! < |xy^2z| < (N+2)!$, si bien que $xy^2z \notin L_2$. Ceci contredit la conclusion du lemme de l'étoile, et montre donc que L_2 n'est pas reconnaissable.

Question 3 • Le développement décimal illimité de $1/7$ est $0{,}142857142857\dots$; donc $\operatorname{Lang}(1/7)$ est décrit par l'expression rationnelle suivante :

$$(142857)^*(\varepsilon + 1 + 14 + 142 + 1428 + 14285)$$

Ceci prouve que ce langage est rationnel. En prime, voici un automate reconnaissant ce langage :



Question 4 • Procédons par l'absurde. Supposons $\operatorname{Lang}(\sqrt{2})$ reconnaissable : le lemme de l'étoile affirme l'existence d'un naturel N tel que tout mot $u \in \operatorname{Lang}(\sqrt{2})$, de longueur au moins égale à N, se décompose en $u = xyz$ avec $y \neq \varepsilon$, $|xy| \leqslant N$ et $xy^nz \in \operatorname{Lang}(\sqrt{2})$ pour tout $n \in \mathbb{N}$. Notons $p = |x|$ et $q = |y|$; alors le mot infini $d(\sqrt{2}) = d_1d_2d_3\dots$ vérifie $d_i = x_i$ pour $1 \leqslant i \leqslant p$, puis $d_j = d_{j+q}$ pour tout $j > p$: pour le voir, il suffit de considérer un préfixe de la forme xy^n suffisamment long pour que $p + nq \geqslant j$. Ainsi, le développement décimal de $\sqrt{2}$ serait périodique à partir d'un certain rang, or ceci est contradictoire puisque $\sqrt{2}$ n'est pas rationnel.

Question 5 • Soient L un langage reconnaissable et $\mathcal{A} = (Q, \delta, i, F)$ un automate fini déterministe reconnaissant L. Dire que L est préfixiel revient à dire que tout préfixe d'un mot de L est encore dans L ; sur l'automate, ceci revient à dire que tous les états traversés lors d'un calcul réussi sont des états d'acceptation. Si l'on a pris le soin d'émonder $\mathcal{A}$, la condition se réduit à $F = Q$, puisque dans ce cas tout état participe à au moins un calcul réussi.

L'algorithme se résume donc comme suit : on émonde l'automate ; on vérifie ensuite qu'il ne reste que des états d'acceptation.

Question 6 • Si $y = \varepsilon$, alors $y^m z = z = y^n z$ quels que soient m et n. On peut donc prendre $m = 2$ et $n = 1$.

Question 7 La suite de terme général $q_k = \delta^*(i, y^k)$ $(k > 0)$ est à valeurs dans l'ensemble fini Q ; il existe donc des exposants m et n distincts, tels que $q_m = q_n$; on peut supposer $m > n > 0$ pour fixer les idées. Alors

$$\begin{aligned} y^m z \in L &\iff \delta^*(i, y^m z) \in F \iff \delta^*\big(\delta^*(i, y^m), z\big) \in F \\ &\iff \delta^*\big(\delta^*(i, y^n), z\big) \in F \iff \delta^*(i, y^n z) \in F \iff y^n z \in L \end{aligned}$$

Ce qui termine la preuve du lemme.

Question 8 • Supposons L_2 reconnaissable et considérons le mot $y = a$. Le lemme de non-pompage assure l'existence de naturels m et n tels que $m > n > 0$ et $y^m z \in L \iff y^n z \in L$ quel que soit le mot z. Prenons $z = a^{m^2+m+1}$: alors $y^m z = a^m a^{m^2+m+1} = a^{m^2+2m+1} = a^{(m+1)^2}$ appartient à L_2. En revanche, $y^n z = a^n a^{m^2+m+1} = a^{m^2+m+n+1}$ n'est pas élément de L_2 ; en effet :

$$m^2 < m^2 + m + n + 1 < m^2 + 2m + 1 = (m+1)^2$$

Par conséquent, $m^2 + m + n + 1$ n'est pas un carré parfait.

Question 9 • Supposons L_3 reconnaissable et considérons le mot $y = ab$. Le lemme de non-pompage assure l'existence de naturels m et n tels que $m > n > 0$ et $y^m z \in L \iff y^n z \in L$ quel que soit le mot z. Prenons $z = (ba)^n a$: alors $y^n z = (ab)^n (ba)^n a$ appartient à L_3, il suffit de prendre $x = (ab)^n$ et $w = a$ pour s'en convaincre. En revanche, $y^m z = (ab)^m (ba)^n a$ n'admet aucune décomposition de la forme $vv^R w$ avec v et w non vides, donc $y^m z \notin L_3$.

Question 10 • La réponse est négative. Soit $N = 4$. Considérons un mot $vv^R w$ de L_3, de longueur au moins égale à N. Comme $v \neq \varepsilon$, on peut écrire $v = au$ où a est une lettre ; deux cas de figure se présentent. Si $u \neq \varepsilon$, prenons $x = \varepsilon$, $y = a$ et $z = uu^R aw$; on a bien $|xy| = 1 \leqslant N$, $y \neq \varepsilon$ et $xy^n z \in L_3$ pour tout $n \in \mathbb{N}$ puisque $xy^0 z = uu^R aw$ (avec $u \neq \varepsilon$ et $aw \neq \varepsilon$) et $xy^n z = aa^R a^{n-2} uu^R aw$ si $n \geqslant 2$ (avec $a \neq \varepsilon$ et $a^{n-2} uu^R aw \neq \varepsilon$). Si $u = \varepsilon$, prenons $x = a^2$, $y = b$ et $z = w'$ où b est la première lettre de w et w' est tel que $w = bw'$; on a bien $|xy| = 3 \leqslant N$, $y \neq \varepsilon$, et $xy^n z \in L_3$ pour tout $n \in \mathbb{N}$ puisque ce mot s'écrit $aa^R y^n z$ avec $a \neq \varepsilon$ et $y^n z \neq \varepsilon$. Ainsi, L_3 vérifie la *conclusion* du lemme de l'étoile ; on ne peut donc pas espérer utiliser celui-ci pour montrer que ce langage n'est pas reconnaissable.

Question 11 • La réponse est négative. Considérons un mot $y \in \mathrm{Lang}(\sqrt{2})$. Comme le développement décimal de $\sqrt{2}$ n'est pas périodique, il existe un exposant $n > 0$ tel que $y^n \notin \mathrm{Lang}(\sqrt{2})$. Alors, comme ce langage est préfixiel, il ne contient pas non plus y^{n+1}. Notons $m = n + 1$; on a $m > n > 0$. Aucun des deux mots $y^m z$ et $y^n z$ n'appartient à $\mathrm{Lang}(\sqrt{2})$ puisque ce dernier est préfixiel. Ainsi, ce langage vérifie la *conclusion* du lemme de non-pompage ; on ne peut donc pas espérer utiliser celui-ci pour montre que ce langage n'est pas reconnaissable.

Références bibliographiques

▶ Référence de l'article de Guo-Qiang Zhang et E. Rodney Canfield présentant le lemme de non-pompage : *The end of pumping*, *Theoretical Computer Science*, vol. 174, Issue 1-2, 1997, 275-279. On trouvera cet article aux URL suivants :

http://www.cs.uga.edu/~gqz/papers/non-pumping.ps.gz
http://www.cs.uga.edu/~gqz/Courses/cs2670/pumping.pdf

▶ Le chapitre 1 du *Handbook of Formal Languages* (éd. Springer), intitulé *Regular languages* et rédigé par Sheng Yu, contient plusieurs variantes du lemme de l'étoile ; en particulier, il propose un lemme énonçant une condition nécessaire et suffisante pour qu'un langage L soit rationnel.

▶ La question 9 provient du chapitre 3 du livre *Introduction to Automata Theory, Languages and Computation*, de John E. Hopcroft et Jeffrey D. Ullman (un grand classique, édité par Addison-Wesley). Cet exercice est marqué d'une «étoile», ce qui indique que les auteurs le considèrent comme plutôt difficile.

Problème 2

Question 1 • L'écriture suivante est récursive terminale, et exploite l'évaluation paresseuse des booléens. Son coût est clairement linéaire en la longueur de la liste, dans le pire des cas.

```
let rec mem x = function
 | [] -> false
 | t::q -> x=t or mem x q ;;
```

Question 2 • Ici, l'écriture n'est pas récursive terminale ; cette propriété pourrait aisément être obtenue, en mettant en œuvre un accumulateur.

```
let rec filtre p = function
 | [] -> []
 | t::q when p t -> t::(filtre p q)
 | _::q -> filtre p q ;;
```

Le coût est proportionnel à la longueur de la liste ℓ, puisque l'on effectue exactement un *Cons* pour chaque élément de cette liste.

Question 3 • L'écriture est immédiate : il suffit de filtrer la liste ℓ_2 avec le prédicat ≪appartenance à la liste ℓ_1≫. Le coût de chaque filtrage est dominé par la longueur de ℓ_1, donc le coût du calcul est dominé par le produit des longueurs des deux listes.

```
let intersection_simple l1 l2 =
 filtre (fun x -> mem x l1) l2 ;;
```

Question 4 • La réponse naïve `let union_simple = prefix @` ne satisfait pas la spécification, puisqu'elle peut introduire des doublons. On va quand même utiliser l'opérateur `@` de concaténation de listes, mais en filtrant l'une des listes, pour ne garder que les éléments qui n'appartiennent pas à l'autre. Le coût est dominé par le produit des longueurs des deux listes, pour la même raison qu'à la question précédente.

```
let union_simple l1 l2 =
 (filtre (fun x -> not (mem x l2)) l1) @ l2 ;;
```

Question 5 • La solution suivante est récursive terminale.

```
let rec forall p = function
 | [] -> true
 | t::q -> (p t) & forall p q ;;
```

Le coût est proportionnel à la longueur de la liste ℓ, puisque l'on effectue exactement un *Cons* pour chaque élément de cette liste. On peut imaginer des formulations plus concises ; en voici deux exemples, l'une utilisant `filtre` et l'autre la fonctionnelle `it_list`. Leur coût reste proportionnel à $|\ell|$.

```
let forall p l = (filtre p l) = l ;;
let forall p l = it_list (prefix &) true (map p l);;
```

Question 6 • La stabilité par complémentation est évidente. Avec la formule $F \cup G = \overline{\overline{F} \cap \overline{G}}$, la stabilité par réunion se réduit à la stabilité par intersection, qu'il reste donc à établir.

• L'intersection d'une partie finie et d'une partie quelconque (cofinie ou non) est finie ; quant à l'intersection de deux parties cofinies $F = \mathbb{N} \setminus P$ et $G = \mathbb{N} \setminus Q$, elle est elle aussi cofinie car $\mathbb{N} \setminus (F \cap G) = P \cup Q$ est finie en tant que réunion de deux parties finies.

• Soit $\mathcal{V}$ une famille de parties de $\mathbb{N}$ contenant les parties finies et stable pour les opérations booléennes : elle est en particulier stable pour la complémentation, donc elle contient les parties cofinies, si bien que $\mathcal{W} \subset \mathcal{V}$.

Question 7 • L'appartenance à une partie finie se teste directement en appliquant `mem` à la liste qui la représente. L'appartenance à une partie cofinie équivaut à la non-appartenance à son complémentaire.

```
let appartient x = function
 | Finie f -> mem x f
 | Cofinie f -> not (mem x f) ;;
```

Question 8 • Revenons à la définition mathématique : $Q \supset P$ ssi tout élément de P appartient à Q. Nous disposons des fonctions `appartient` et `forall`. Si Q est finie, on vérifie que tous ses éléments appartiennent à P ; si P et Q sont cofinies, on vérifie que $\overline{Q}$ contient $\overline{P}$; enfin, si$P$ est finie et Q cofinie, la réponse est `false` ûisqu'une partie finie ne peut pas contenir une partie cofinie !

```
let rec contient p q = match (p,q) with
 | (_,Finie qq) -> forall (fun x -> appartient x p) qq
 | (Cofinie pp,Cofinie qq) -> contient (Finie qq) (Finie pp)
 | (Finie _,Cofinie _) -> false ;;
```

Question 9 • L'écriture est immédiate, avec la structure de données choisie.

```
let complement = function
 | Finie f -> Cofinie f
 | Cofinie f -> Finie f ;;
```

Question 10 • Si les deux parties sont finies, il suffit d'utiliser `union_simple`. Sinon, la réunion est une partie cofinie ; le cas de la réunion de deux parties cofinies est simple, avec la relation :

$$\big(\mathbb{N} \setminus P\big) \cup \big(\mathbb{N} \setminus Q\big) = \mathbb{N} \setminus \big(P \cap Q\big)$$

Il reste le cas de la réunion d'une partie finie P et d'une partie cofinie $\mathbb{N} \setminus Q$. Le résultat est une partie cofinie $\mathbb{N} \setminus R$. On a :

$$R = \mathbb{N} \setminus \big(P \cup (\mathbb{N} \setminus Q)\big) = (\mathbb{N} \setminus P) \cap Q$$

Donc R est l'ensemble des éléments de Q qui *n'appartiennent pas* à P ; le calcul de cette partie se fait avec `filtre`.

```
let union f g = match (f,g) with
 | (Finie p,Finie q) -> Finie (union_simple p q)
 | (Cofinie p,Cofinie q) -> Cofinie (intersection_simple p q)
 | (Finie p,Cofinie q) -> let fp = fun x -> not (mem x p)
    in Cofinie (filtre fp q)
 | (Cofinie p,Finie q) -> let fq = fun x -> not (mem x q)
    in Cofinie (filtre fq p) ;;
```

Question 11 • L'écriture est immédiate si l'on veut bien se souvenir cette fois que $F \cap G = \overline{\overline{F} \cup \overline{G}}$.

```
let intersection f g =
 let f1 = complement f and g1 = complement g
 in complement (union f1 g1) ;;
```

Question 12 • Le test d'appartenance a un coût égal à la longueur de la liste, dans le pire des cas. Le test d'inclusion est réalisé par application à tous les éléments d'une liste du test d'appartenance à l'autre liste ; donc son coût est égal au produit des longueurs des deux listes dans le pire des cas. La complémentation a un coût nul (toujours en nombre de *Cons*). La réunion et l'intersection ont elles aussi un coût au moins égal au produit des longueurs des deux listes dans le pire des cas, puisque l'on applique à tous les éléments d'une liste le test d'appartenance ou de non-appartenance à l'autre liste.

Question 13 • En changeant de structure de données pour représenter les parties finies ou cofinies de $\mathbb{N}$, on peut diminuer le coût de la recherche d'un élément. Avec des arbres binaires de recherche, ce coût passera de n à $\ln n$ (en moyenne), où n est le nombre d'éléments dans la structure ; si l'on utilise des arbres binaires *équilibrés*, le coût sera un $\mathcal{O}(\ln n)$ dans le pire des cas. Enfin, en faisant appel à une table de hachage, le coût moyen devient constant.

Question 14 • Il suffit de redéfinir le type `ficof` en le paramétrant :

```
type 'a ficof = Finie of 'a list | Cofinie of 'a list ;;
```

Le polymorphisme de Caml fera le reste ! Notons que le type de `appartient` deviendra `'a -> ficof -> bool`. Évidemment, on ne saurait avoir le beurre et l'argent du beurre : on ne pourra implanter une structure de données performante que si le type de base le permet. En particulier, l'emploi d'arbres binaires de recherche n'est possible que si ce type est muni d'une relation d'ordre total.

FIN