Mines 2009 - Informatique
Un corrigé.

1 Automates.

1. On note $|z|$ la longueur d'unmot z. Montrons que

$$(\forall v \in \Sigma^*,\ \phi(uv) = \phi(u)\phi(v)) \iff |u| = 0[2]$$

   - Le sens réciproque est immédiat par définition de ϕ.
   - Pour prouver le sens direct, on contrapose. On suppose donc que u est un mot de longueur impaire $2k+1$. u s'écrit $u = u_1 \dots u_{2k+1}$. Soit v le mot constitué d'une unique lettre v_1 avec $v_1 \neq u_{2k+1}$. $\phi(uv)$ se termine par u_{2k+1} at $\phi(u)\phi(v)$ se termine par v_1 et ces deux mots sont donc différents.

2. Les mots u de longueur paire tels que $\phi(u) = u$ sont ceux du type $l_1 l_1 \dots l_k l_k$ et le mot vide. Ceux de longueur impaire sont ceux du type $l_1 l_1 \dots l_k l_k l_{k+1}$ (éventuellement réduits à une lettre).
   Les éléments de L_1 sont donc les mots de longueur plus grande que 2 et où l'une au moins des lettre d'indice pair est différente de la la lettre d'indice impair qui la précède. Ce sont donc les mots du type

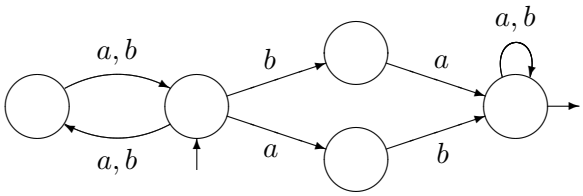
$$vabw \ \text{ ou } \ vbaw$$

   où v est un mot de longueur paire, w est un mot quelconque.

3. On a ainsi directement

$$L_1 = (ab + aa + ba + aa)^*(ab + ba)(a + b)^* = (\Sigma^2)^*(ab + ba)\Sigma^*$$

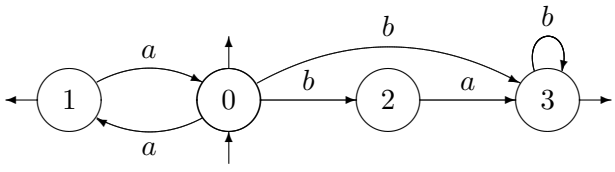
4. L'automate suivant convient donc



5. Un élément de $\phi(L_2)$ est l'image par ϕ d'un mot du type $a^n b^p$. Si $n = 2k$ est pair, cette image est $a^{2k}b^p$. Si $n = 2k+1$, elle vaut $a^{2k}bab^{p-1}$ si $p \geq 1$ et a^{2k+1} si $p = 0$. On a donc

$$\phi(L_2) = (a^2)^*b^* + (a^2)^*abb^* + (a^2)^*a = (a^2)^*(b^* + bab^* + a)$$

6. Je propose l'automate suivant :



   - On est dans l'état 0 après lecture d'un élément de $(a^2)^*$.
   - On est dans l'état 1 après lecture d'un élément de $(a^2)^*a$.
   - On est dans l'état 2 après lecture d'un élément de $(a^2)^*b$.
   - On est dans l'état 3 après lecture d'un élément de $(a^2)^*bb^*$ ou de $(a^2)^*bab^*$.

   Les états terminaux étant $0, 1$ et 3, le langage reconnu correspond bien à l'ensemble des mots décrits à la question précédente.

7. On procède par induction structurelle. La propriété à prouver est

$$H_L \ : \ P(L) \ \text{ et } \ I(L) \ \text{ sont rationnels}$$

et on veut montrer que H_L est vraie pour tout langage L rationnel.
- Initialisation : la propriété est immédiatement vraie si L est le langage vide (on a alors $P(L) = I(L) = \emptyset$) ou un langage contenant un unique mot d'une lettre (on a alors $P(L) = \emptyset$ et $I(L) = L$).
- Hérédité : soient L et M deux langages rationnels pour lesquels la propriété est vraie. Alors

$$P(L+M) = P(L) + P(M) \ \text{ et } \ I(L+M) = I(L) + I(M)$$

et, avec l'hypothèse de récurrence, l'hypothèse est vraie pour $L+M$. De plus

$$P(LM) = P(L)P(M) + I(L)I(M) \ \text{ et } \ I(LM) = P(L)I(M) + I(L)P(M)$$

et H_{LM} est vraie. Enfin

$$P(L^*) = P(L)^*(I(L)P(L)^*I(L)P(L)^*)^* \ \text{ et } \ I(L^*) = P(L^*)I(L)P(L^*)$$

et H_{L^*} est vraie.

8. Soit q un état utile de A. Il existe un chemin de l'état à q étiqueté par un certain u et un autre de q à un état final étiqueté par un certain mot v. Si, par l'absurde, il existe une transition (q, x, q) dans A alors les mots uv et uxv sont tous deux reconnus et l'un des deux est de longueur impaire, ce qui est exclus.
9. Soit q un état utile de A. Il existe un chemin de l'état à q étiqueté par un certain u et un autre de q à un état final étiqueté par un certain mot v. Soit w un mot étiquetant un chemin de l'état initial à q. ALors uv et wv sont reconnus et sont donc de longueur paire. Ainsi $|u| + |v| = |w| + |v|[2]$ et donc $|u| = |w|[2]$ ce qui signifie que longueurs de u et v ont même parité.
10. On doit prouver une double inclusion.
    - Soit u un mot reconnu par A. Il existe un calcul réussi dans A étiqueté par u. Si ce calcul ne passe pas par q alors c'est aussi un calcul réussi dans $S(A, q)$ (c'est le cas si $u = \varepsilon$). Sinon, il contient des sous-chemins $q' \xrightarrow{x} q \xrightarrow{y} q''$ (q n'est ni initial ni terminal) et $q', q'' \neq q$. Par construction, il existe dans $S(A, q)$ un chemin de longueur 2 d'étiquette xy passant de q' à q''. En faisant les substitutions, on obtient dans $S(A, q)$ un chemin étiqueté par u. Dans tous les cas, u est reconnu par $S(A, q)$.
    - Réciproquement, soit u un mot reconnus par $S(A, q)$. Il existe un calcul réussi dans $S(A, q)$ étiqueté par u. Si ce calcul ne passe que par des états de A alors c'est un calcul réussi dans A (c'est le cas si $u = \varepsilon$ car q n'était pas initial dans A). Sinon, il existe des sous-chemins $q' \xrightarrow{x} r \xrightarrow{y} q''$ avec r qui n'est pas un état de Q (un tel état n'apparaît pas en début ou bout de chaîne par construction) et q', q'' qui en sont. Par construction, il existe un calcul similaire dans A (de q' à q'' étiqueté par xy). En faisant les substitutions, on obtient dans A un chemin étiqueté par u. Dans tous les cas, u est reconnu par A.
11. Soit L un langage rationnel. $P(L)$ est alors rationnel et est reconnu par un automate A. Ce dernier vérifie les propriétés vues en questions 8 et 9 et, en particulier il n'existe aucune transition d'un état utile vers lui même.
    On peut itérer le processus décrit en question 10 pour obtenir un automate A' qui reconnaît encore $P(L)$ et qui ne contient aucun état utile qui ne soit ni initial ni final et duquel partent ou arrivent deux transitions (le processus est effectué un nombre fini de fois car seuls les états initialement présents dans A peuvent violer la propriété ; dans la construction, on n'ajoute aucun état violant la propriété et on en supprime un). Pour chaque état r de A' non présent dans A, on sait qu'il existe un unique chemin de A' du type $q' \xrightarrow{x} r \xrightarrow{y} q''$. On construit A_P à partir de A' de la façon suivante en changeant les transitions précédentes en $q' \xrightarrow{y} r \xrightarrow{x} q''$. Il est alors "quasi-immédiat" que A_P reconnaît $\phi(P(L))$. Ce dernier langage est donc rationnel.

12. On procède par récurrence de construction. Il s'agit de montrer que l'hypothèse

$$H_L \ : \ \forall x \in \Sigma, \ M(L,x) \ \text{ est rationnel}$$

est vraie pour tout langage rationnel L.
- Initialisation : la propriété est immédiatement vraie si L est le langage vide (on a alors $M(L,x) = \emptyset$) ou un langage contenant un unique mot d'une lettre (on a alors $M(L,x) = \emptyset$ ou $M(L,x) = \{\varepsilon\}$).
- Hérédité : soient L et L' deux langages rationnels pour lesquels la propriété est vraie. Pour tout $x \in \Sigma$, on a

$$M(L+L',x) = M(L,x) + M(L',x)$$

$$M(LL',x) = \begin{cases} LM(L',x) & \text{si } \ \varepsilon \notin L' \\ M(L,x) + LM(L',x) & \text{sinon} \end{cases}$$

$$M(L^*) = L^* M(L,x)$$

La propriété est donc vraie pou $L+L'$, LL' et L^*.

13. Un mot u dans $I(L)$ s'écrit wa ou wb et on a $\phi(u)$ qui vaut $\phi(w)a$ ou $\phi(w)b$. Dans le premier cas, w est dans $M(I(L),a)$ et dans le second il est dans $M(I(L),b)$. Finalement, tout mot de $\phi(I(L))$ est dans $\phi(M(I(L),a))a$ ou $\phi(M(I(L),b))b$.
Réciproquement, un mot de cet ensemble s'écrit $\phi(w)a$ (ou $\phi(w)b$) avec w dans $M(I(L),a)$ (ou $M(I(L),b)$). On a alors $u = wa$ (ou $u = wb$) dans $I(L)$ et $\phi(u) = \phi(w)a$ (ou $\phi(w)b$). Ainsi, notre mot est dans $\phi(I(L))$.

$$\phi(I(L)) = \phi(M(I(L),a))a + \phi(M(I(L),b))b$$

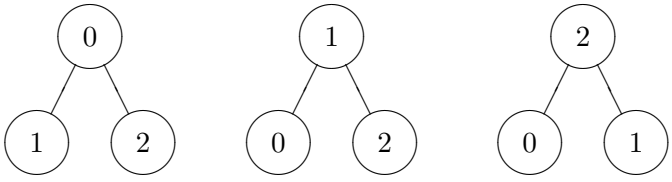
14. $\phi(L) = \phi(P(L)) + \phi(I(L))$.
$\phi(P(L))$ est rationnel avec la question 11. $L' = M(I(L),x)$ l'est avec les questions 7 et 12. De plus, $L' = P(L')$ et donc $\phi(L') = \phi(P(L'))$ l'est aussi. La question 13 donne ainsi que $\phi(I(L))$ est rationnel.
$\phi(L)$ est ainsi rationnel comme réunion de tels langages.
15. Si $\Phi(L)$ est rationnel alors $L = \phi(\phi(L))$ l'est. En contraposant, si L est non rationnel, $\phi(L)$ est non rationnel.
16. Il suffit d'inverser les lettres par couples.

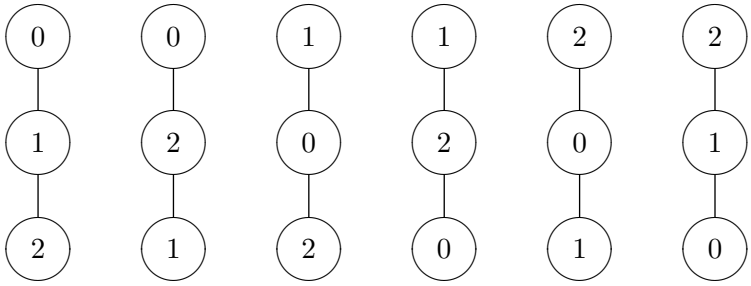
```
let rec phi u =
   match u with
   [] -> []
   [x] -> [x]
 | x::y::r->y::x::(phi r) ;;
```

2 Algorithmique.

2.1 Du codage racine-fils-frères au codage de Prüfer.

17. La racine d'un arbre à trois noeuds a 1 ou 2 fils (deux "squelettes" possibles). Pour chaque squelette, différentes numérotations sont possibles.





18. On initialise un tableau de bonne dimension contenant la valeur -1. On écrit une fonction (récursive) locale `affecte` prenant en argument deux entiers `r` et `i` tels que `i` est soit égal à -1 soit égal à l'un des fils de `r`. La fonction agit comme suit :
    - elle ne fait rien si `i` vaut -1
    - sinon, elle remplit le tableau des pères pour `i`, ses frères situés à droite et tous leurs descendants.

    Ainsi, un appel initial avec la racine et son fils le plus à gauche remplira tout le tableau des pères (sauf la case de la racine mais elle contient déjà la bonne valeur -1).

```
let calculer_peres racine fils freres =
  let n=vect_length fils in
  let peres=make_vect n (-1) in
  let rec affecte r i =
    if i<>(-1) then begin
              peres.(i) <- r ;
              affecte r freres.(i) ;  (* on passe au fr\‘ere \‘a droite *)
              affecte i fils.(i)   (* on passe au premier fils *)
              end
   in affecte racine fils.(racine);peres ;;
```

19. Chaque case du tableau des pères est remplie une unique fois (après l'initialisation) et à chaque remplissage est associé un nombre constant d'opérations. La complexité de la fonction est donc linéaire en fonction du nombre de noeuds de l'arbre.

20. Pour calculer l'arités d'un noeud donné i, on regarde son fils et, s'il existe, on compte ses frères successifs. On fait cela pour chaque noeud i.

```
let calculer_arites fils freres =
  let n=vect_length fils in
  let arites=make_vect n 0 in
  for i=0 to n-1 do
     let j=ref fils.(i) in
     while !j<>(-1) do
           arites.(i) <- arites.(i)+1 ;
           j:= freres.(!j)
           done ;
     done ;
  arites ;;
```

21. i étant fixé, le remplissage de la case numéro i du tableau des arités a un coût proportionnel au nombre des fils de i plus 1. Le coût global de la fonction est donc proportionnel à

$$\sum_{i=0}^{n-1}(f_i+1)$$

où f_i est le nombre de fils du noeud i. Or, chaque noeud (hormis la racine) a un unique père et est compté une unique fois comme fils d'un autre noeud. Ceci signifie que $\sum_{i=0}^{n-1} f_i = n-1$. Finalement, le coût de la fonction `calculer_arites` est linéaire en fonction du nombre de noeuds de l'arbre.

22. On incrémente une référence i correspondant à un numéro de case jusqu'à trouver une valeur strictement plus petite que d (ou à atteindre la fin de la partie utile du tableau). On doit insérer d en position i atteinte. Avant cela, on décale toutes les case de numéro $\geq i$ d'une position vers la droite (en commençant par la droite pour ne pas perdre de valeur).

```
let inserer table nb d =
let i=ref 0 in
while !i<nb & table.(!i)>=d do incr i done ;
for k=nb downto !i+1 do table.(k)<-table.(k-1) done ;
table.(!i)<-d ;
nb+1 ;;
```

23. On parourt une unique fois les cases "utiles" du tableau (pour comparer à d ou pour déplacer le contenu). La complexité est donc $O(nb)$.

24. Le codage de Prüfer de A_3 est

$$9, 1, 6, 7, 1, 3, 7, 9, 9, 3$$

25. Dans la partie préparatoire, on gère une référence `nb` indiquant le nombre de feuilles rencontrées. C'est le nombre de cases utiles du tableau `table` évoqué par l'énoncé (qui contient les numéros des feuilles par ordre décroissant).
Pour construire le codage de Prüfer, on remplit les $n-1$ cases d'un tableau à l'aide d'une boucle. A l'étape i, on prend la feuille de numéro minimal (l'élément numéro `nb`-1 de `table`) et on remplit la case numéro i avec le numéro de son père. Il faut alors penser à mettre à jour `table` (il y a une feuille en moins mais aussi peut-être une autre à ajouter si l'élément que l'on vient de mettre en position numéro i devient une feuille) et le tableau des arités (un noeud vient de perdre un fils).

```
let calculer_Prufer racine fils freres =
  let n=vect_length fils in
        (* partie pr\'eparatoire *)
  let arites=calculer_arites fils freres in
  let peres=calculer_peres racine fils freres in
  let table=make_vect n (-1) in
  let nb=ref 0 in
  for i=0 to n-1 do
     if fils.(i) = -1 then nb:=inserer table !nb i done ;
        (* calcul du codage de Prufer *)
  let prufer=make_vect (n-1) (-1) in
  for i=0 to n-2 do
    prufer.(i)<-peres.(table.(!nb-1)) ;
    decr nb ;  (* on vient d'enlever une feuille *)
    if arites.(prufer.(i)) = 1 then nb:=inserer table !nb prufer.(i) ;
    arites.(prufer.(i)) <- arites.(prufer.(i)) -1
    done ;
  prufer ;;
```

26. Le calcul des tableaux `arites` et `peres` se fait en $O(n)$ ainsi que les initialisations. Le coût global de la gestion de `table` est au pire en $O(n^2)$ (chaque élément est inséré une unique fois et chaque insertion a un coût majoré par $O(n)$). Pour les reste, le coût est $O(n)$ (hors la gestion de `table` chaque étape de la boucle se fait en temps constant). Ainsi, le coût de notre fonction est au pire

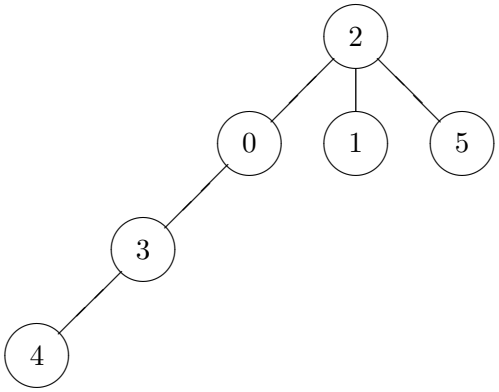
quadratique en fonction de n.
Avec la fonction écrite, il y a un cas où cette complexité est atteinte (si la racine possède $n-1$ fils alors le remplissage initial de `table` se fait en $O(n^2)$).
Remarque : en faisant un premier passage pour compter le nombre de feuilles puis un second pour remplir `table` *on peut améliorer l'initialisation de ce tableau et atteindre une complexité linéaire pour celle-ci. Le cas le pire évoqué n'en est alors plus un. Nous ne chercherons pas s'il en existe un autre...*

2.2 Du codage de Prüfer à un codage racine-fils-frères.

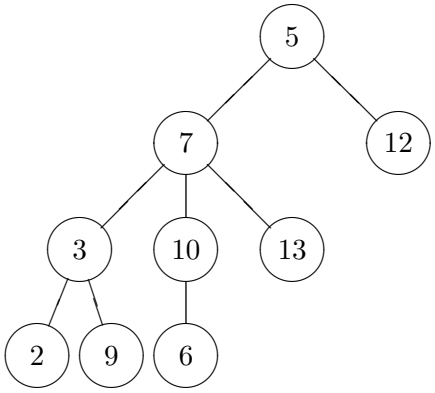
27. Par construction, le nombre de fils d'un noeud (son arité) est égal au nombre de fois où il apparaît dans le codage de Prüfer.

```
let calculer_arites_par_Prufer prufer =
  let n=vect_length prufer in
  let arites=make_vect (n+1) 0 in
  for i=0 to n-1 do arites.(prufer.(i)) <- arites.(prufer.(i)) + 1 done;
  arites ;;
```

28. Les noeuds sont numérotés de 0 à 5. Le tableau des arités est $[|1;0;3;1;0;0|]$. Les feuilles sont donc $1, 4$ et 5. La plus petite est 1 et, avec le codage de Prufer, son père est 2. 1 étant supprimé, 2 a encore deux fils et il reste 4 et 5 comme feuilles. La plus petite est 4 et son père est trois. Quand on supprime la feuille 4, 3 devient une feuille et on dispose des feuilles 3 et 5. La plus petite est 3 et son père est 0. Quand on supprime 3, 0 devient une feuille. A l'étape suivante on obtient le père de 0 qui est 2 puis le père de 5 qui est 2. On a alors l'arbre suivant.



29. Les feuilles de l'arbre sont $1, 6, 9, 12$ et 13 (les numéros qui ne sont pas dans le codage de Prüfer). La première étape de ce codage est de supprimer 1 et on sait que son père est 3. Cette suppression ne donne pas de nouvelle feuille.
A l'étape suivante, on supprime 6 dont le père est 10 et 10 devient feuille. On continue à l'avenant pour obtenir l'arbre suivant.



30. Comme pour la fonction `calculer_Prufer`, dans une phase préparatoire, on initialise les tableaux `fils` et `freres` à la valeurs -1 (on saura ainsi quand un fils a déjà été attribués), on calcule le tableau `arites` et on gère un tableau `table` des feuilles disponibles (par ordre décroissant) et une référence `nb` en donnant le nombre.
Comme il est expliqué dans les questions précédentes, on traite au fur et à mesure la "feuille disponible de numéro minimal" pour laquelle le père est connu (avec le tableau `prufer`) : cela permet de remplir une case de `fils` ou de `freres` (si le noeud dont on a trouvé a un fils a déjà un fils). Il faut bien sûr penser à mettre à jour `nb`, `table` et `arites` en conséquence.

```
let calculer_arbre prufer fils freres =
  let n=vect_length fils in
     (* \'etape pr\'eparatoire *)
  let arites=calculer_arites_par_Prufer prufer in
  let table=make_vect n (-1) in
  let nb=ref 0 in
  for i=0 to n-1 do fils.(i) <- -1 ; freres.(i) <- -1 done ;
  for i=0 to n-1 do
     if arites.(i) = 0 then nb:=inserer table !nb i done ;
     (* remplissage des tableaux *)
  for i=0 to n-2 do   (
     let k = table.(!nb-1) in  (* feuille de num\'ero minimal de pere pruf.(i) *)
     if fils.(pruf.(i)) = -1 then fils.(pruf.(i)) <- k
     else begin  (* cas o\'u pruf.(i) a d\'ej\'a un fils *)
          let j=ref (fils.(pruf.(i))) in
          while freres.(!j)<>(-1) do j:=freres.(!j) done ;
          freres.(!j) <- k
          end ;
     nb:= !nb-1 ;
     if arites.(pruf.(i)) = 1 then nb:=inserer table !nb pruf.(i) ;
     arites.(pruf.(i)) <- arites.(pruf.(i)) -1
     done ;
  table.(0) ;;
```

31. Pr est une application de $\mathcal{A}(E)$ dans $\mathcal{S}(E)$. Montrons par récurrence sur le cardinal de E que c'est une bijection.
- Si $n = 1$ et $E = \{p\}$ alors il y a un unique arbre et une unique suite. On a bien une bijection.
- Soit $n \geq 2$ tel que le résultat soit vrai jusqu'au rang $n-1$. Soit $E = \{x_1, \ldots, x_n\}$ un ensemble de cardinal n.
  Supposons que deux arbres $a, b \in \mathcal{A}(E)$ aient même codage de Prüfer P. Au moins un élément de E n'est pas dans P et par construction, le plus petit de ces éléments x est la feuille de numéro minimal de a et de b. Les arbre a' et b' obtenus au supprimant cette feuille sont éléments de $\mathcal{A}(E \setminus \{x\})$ et ont même codage de Prüfer et sont donc égaux par hypothèse de récurrence. Comme x a le même père dans a et b (le premier élément de P), les arbres a et b sont égaux. On a donc l'injectivité de Pr.
  Soit $P \in \mathcal{S}(E)$; au moins un élément de E n'est pas dans P et on peut noter x le plus petit de ces éléments. Soient p le premier élément de P et P' la suite des éléments restant. $P' \in \mathcal{S}(E \setminus \{x\})$ et il existe (récurrence) $a' \in \mathcal{A}(E \setminus \{x\})$ dont le codage de Prüfer soit P'. On adjoint au noeud p de P' le fils x. On obtient $a \in \mathcal{A}(E)$ tel que $Pr(a) = P$. On a donc la surjectivité de Pr.

Remarque : les procédés récurrents décrits sont les versions récursives des algorithmes itératifs donnés plus haut.

32. Il y a n^{n-1} éléments dans $\mathcal{S}(E)$ (n choix pour chaque élément de la suite et l'ordre importe) et

ainsi

$$\mathrm{card}(\mathcal{A}(E)) = n^{n-1}$$

Remarque : c'est cohérent avec le nombre d'arbres consécutifs trouvés dans le cas $n = 3$.