Mines-Ponts 2009. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

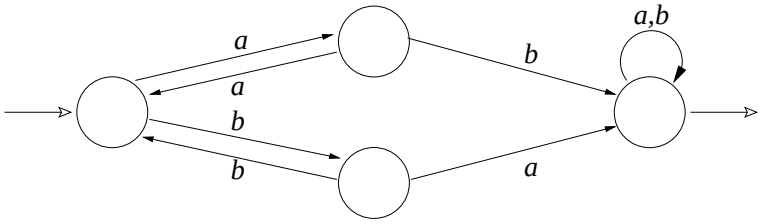
Partie I. Automates.

1) Par définition même de ϕ, il est immédiat qu'une condition suffisante est que u soit de longueur paire.
Elle est également nécessaire car si $u = u_1u_2\dots u_{2k+1}$ et si $v = v_1$ avec $v_1 \neq u_{2k+1}$ (bien possible puisque $\operatorname{card}\Sigma \neq 1$) alors les deux dernières lettres de $\phi(uv)$ et de $\phi(u)\phi(v)$ sont différentes. □

2) $u \in L_1$ si et seulement si il existe un entier k telque $2 \leqslant 2k \leqslant |u|$ et $u_{2k} \neq u_{2k-1}$. □

3) Notons k le plus petit indice tel que $u_{2k} \neq u_{2k-1}$. Alors u est de la forme $uabw$ ou $ubaw$ avec u un mot de longueur paire tel que chaque lettre d'indice impair soit égale à la suivante et w un mot quelconque.
Ainsi $L_1 = (aa + bb)^*(ab + ba)(a + b)^*$. □

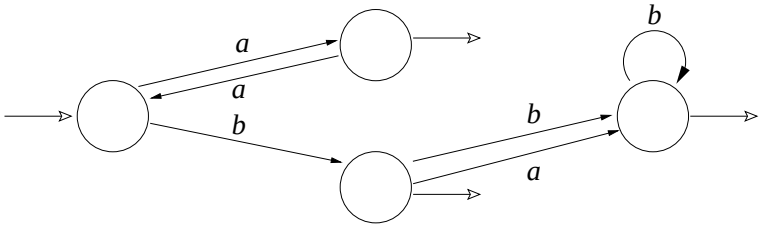
4)



Un automate reconnaissant L_1

5) Un mot appartenant à L_2 est de la forme $u = a^pb^q$. Si $p = 2r$ alors $\phi(u) = u = a^{2r}b^q$ et si $p = 2r + 1$ alors $\phi(u) = u = a^{2r+1}$ si $q = 0$ et $\phi(u) = a^{2r}bab^{q-1}$ si $q \neq 0$.
Une expression rationnelle reconnaissant $\phi(L_2)$ est donc $(aa)^*(a + b^* + bab^*)$. □

6)



Un automate reconnaissant $\phi(L_2)$

7) Notons P (resp. I) le langage constitué des mots de longueur paire (resp. impaire) sur l'alphabet Σ. Ce langage est rationnel car décrit par l'expression $(\Sigma\Sigma)^*$ (resp. $(\Sigma\Sigma)^*\Sigma$).
Or l'intersection de deux langages rationnels est rationnel (en effet une facile conséquence du théorème de Kleene est que si un langage est régulier alors son complémentaire l'est aussi et il suffit de remarquer que $L_1 \cap L_2 = \overline{\overline{L_1} + \overline{L_2}}$).
Ainsi $P(L) = L \cap P$ et $I(L) = L \cap I$ sont rationnels. □

8) Soit q un état utile d'un tel automate, c'est à dire tel qu'il existe un chemin réussi passant par q. Il existe donc un mot u faissant passr de l'état initial à q et un mot v faisant passer de q à un état final. Ainsi le mot uv est reconnu donc de longueur paire.
S'il existait une transition (q, x, q) alors le mot uxv serait reconnu ce qui est impossible car il est de longueur impaire.□

9) Avec les notations de la question précédente soient deux mots u_1 et u_2 faisant passer de q_0 à q. Alors u_1v et u_2v sont reconnus donc de longueur paire donc u_1 et u_2 ont même parité. □

10)

a) Soit u un mot reconnu par A.
Si le chemin correspondant ne passe pas par q alors le même chemin figure dans $S(A, q)$ donc u est également reconnu par $S(A, q)$.

Sinon par hypothèse (car q n'est ni initial ni final) le chemin contient un sous-chemin $q' \xrightarrow{x} q \xrightarrow{y} q''$ et par construction $S(A,q)$ contient un sous chemin $q' \xrightarrow{x} r \xrightarrow{y} q''$ donc reconnait également le mot u car le complémentaire du sous-chemin figure à la fois dans A et dans $S(A,q)$ puisque q' et q'' sont différents de q (en effet aucune transition de A n'admet q à la fois comme origine et extrémité).

b) Réciproquement soit un mot u reconnu par $S(A,q)$.
Si le chemin correspondant ne passe que par des états de A alors il figure aussi dans A qui reconnait ainsi le mot u.
Sinon il contient un sous-chemin $q' \xrightarrow{x} r \xrightarrow{y} q''$ avec r n'appartenant pas à A alors que q' et q'' y appartiennent. Par construction A contient le sous-chemin $q' \xrightarrow{x} q \xrightarrow{y} q''$ et reconnait donc u.

c) En conclusion les automates A et $S(A,q)$ reconnaissent le même langage. □

11) Soit L un langage rationnel. D'après la question 7), le langage $P(L)$ est également rationnel. D'après le théorème de Kleene il existe un automate fini le reconnaissant. D'après la question 8) si q est un état utile, il n'existe aucune transition d'origine et d'extrémité q. On peut donc appliquer de manière itérée la construction de la question 10) pour chaque état utile qui soit extrémité ou origine d'au moins deux transitions.
On obtient ainsi un automate A reconnaissant $P(L)$ tel que chaque état utile non initial ou final soit origine d'une unique transition et extrémité d'une unique transition.
Soit un état utile q (non initial ni final) de A. Il existe un unique chemin réussi passant par q : $q_0 q_1 \dots q_{2n}$ correspondant au mot $u_1 u_2 \dots u_{2n}$ et tel que $q = q_k$ avec $0 < k < 2n$.
Soit alors l'automate A' obtenu à partir de A de la manière suivante :
- si k est pair on remplace la transition $q_k \xrightarrow{u_{k+1}} q_{k+1}$ par $q_k \xrightarrow{u_{k+2}} q_{k+1}$
- si k est impair on remplace la transition $q_k \xrightarrow{u_{k+1}} q_{k+1}$ par $q_k \xrightarrow{u_k} q_{k+1}$
- on remplace la transition $q_0 \xrightarrow{u_1} q_1$ par $q_0 \xrightarrow{u_2} q_1$
En somme cela revient à appliquer la transformation ϕ à l'étiquette généralisée de tout chemin réussi de A.

Il est alors immédiat qu'un mot u est reconnu par A' si et seulement si $\phi(u)$ est reconnu par A donc si et seulement si $\phi(u) \in P(L)$ donc si et seulement si $u \in \phi((PL))$ car ϕ est involutive.
D'après le théorème de Kleene le langage $\phi\Big(P(L)\Big)$ est rationnel. □

12) Soit A un automate reconnaissant L et soit A' l'automate obtenu à partir de A en conservant les mêmes états et transitions mais en décrétant comme état final tout état q tel qu'il existe une transition $q \xrightarrow{x} q'$ où q' est un état final de A.
Par construction l'automate A' reconnait le langage $M(L,x)$ qui de ce fait est rationnel. □

13) Soit $u \in I(L)$. Alors u s'écrit va (resp. vb) avec $v \in M(I(L),a)$ (resp. $v \in M(I(L),b)$) et $\phi(u) = \phi(v)a$ (resp. $\phi(v)b$) car v est de longueur paire.
Ainsi $\phi(u) \in \phi\Big(M(I(L),a)\Big)a$ (resp $\phi(u) \in \phi\Big(M(I(L),b)\Big)b$).
Réciproquement si $w \in \phi\Big(M(I(L),a)\Big)a$ par exemple alors $w = \phi(v)a$ avec $v \in M(I(L),a)$ donc v de longueur paire. Alors $u = va \in I(L)$ et $w = \phi(v)a = \phi(va) = \phi(u)$ appartient bien à $\phi(I(L))$.

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

14) Soit L un langage rationnel.

a) Alors $I(L)$ est rationnel (question 7)) donc $M(I(L),x)$ également par la question 12). Donc, puisque ce langage ne comporte que des mots de longueur paire, par la question 11), le langage $\phi\Big(M(I(L),x)\Big)$ est également rationnel.

Il en découle que $\phi\Big(I(L)\Big)$ est rationnel par la question 13).

b) Par la question 11) la langage $\phi\Big(P(L)\Big)$ est rationnel.

c) Ainsi $\phi(L) = \phi\Big(P(L)\Big) + \phi\Big(I(L)\Big)$ est rationnel. □

15) Si $\phi(L)$ est rationnel alors, par la question précédente, $L = \phi\big(\phi(L)\big)$ également. □

16)

```
let rec phi u = match u with
  | [] -> []
  | [x] -> [x]
  | x :: suite -> (hd suite) :: x :: phi(tl suite)
;;
phi : 'a list -> 'a list = <fun>
```

Partie II. Algorithmique.

Première partie : du codage racine-fils-frères au codage de Prüfer.

17) Il y a $3! = 6$ arbres "linéaires" et 3 de hauteur 1.

18) Le principe est simple : on parcourt une fois le tableau $fils$ par une boucle for. À l'itération i, si i n'est pas une feuille, on détermine ainsi le père (à savoir i) de son fils j le plus à gauche puis à l'aide du tableau $freres$ et d'une boucle *while* on détermine le père (toujours i) des frères éventuels de j.
On initialise naturellement le tableau *peres* par des -1 de sorte que à la fin la seule case non remplie à savoir celle de la racine contiendra bien -1. Ce qui montre d'ailleurs qu'il est inutile de passer *racine* en argument de la fonction.

```
let calculer_peres fils freres =
  let n = vect_length fils in let peres = make_vect n (-1) in
    for i=0 to (n-1) do
      if (fils.(i) <> -1) then let j = ref fils.(i) in
        while (!j <> -1) do
          peres.(!j) <- i;
          j := freres.(!j)
        done
    done;
    peres
;;
calculer_peres : int vect -> int vect -> int vect = <fun>
```

19) Le coût de l'itération i est celui d'une comparaison ($fils(i) \neq -1$)) puis si i n'est pas une feuille celui du parcours de tous ses fils qui est proportionnel à son nombre r_i (deux affectations et une comparaison par fils). En notant r_i le nombre éventuellement nul de fils de i, la complexité est donc de $\alpha n + \beta \sum_{i=0}^{n-1} r_i = (\alpha + \beta)n$ car $\sum_{i=0}^{n-1} r_i = n$.
La complexité est donc linéaire. □

20) Le principe est semblable en commençant par remplir le tableau arite par des 0.

```
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
      if (fils.(i) <> -1) then let j = ref fils.(i) in
        while (!j <> -1) do
          arites.(i) <- succ arites.(i);
          j := freres.(!j)
        done;
    done;
    arites
;;
calculer_arites : int vect -> int vect -> int vect = <fun>
```

21) Par un raisonnement en tout point analogue à la question 19), la complexité est linéaire. □

22)

```
let inserer table nb d = let i = ref(0) in
  while (!i < nb) & (table.(!i) >= d) do incr i done;
  for j=nb downto (!i+1) do table.(j) <- table.(j-1) done;
  table.(!i) <- d;
  nb + 1
;;
inserer : 'a vect -> int -> 'a -> int = <fun>
```

23) Si on note i la position d'insertion, la complexité est de $i + 1$ comparaisons et $nb - i + 1$ affectations. On peut retenir que la complexité est linéaire par rapport à nb.

25) À l'aide du tableau $fils$ lu en sens inverse, on commence par créer facilement un tableau $feuilles$ de taille n dont les nb premières cases contiennent les feuilles triées par ordre d'étiquette décroissante de l'arbre initial.
On remplit alors le tableau $Prufer$ par une boucle for de 0 à $n-2$ en opérant ainsi à chaque itération :
- on range le père de $feuilles(nb)$ dans $Prufer(i)$ puis on maintient à jour le tableau $feuilles$ en décrémentant nb puis
- si $feuille(nb)$ est fils unique, on insère son père dans le tableau $feuilles$ (ce qui réincrémente nb qui retrouve sa valeur du tour précédent)
- sinon on ne touche pas le tableau $feuilles$ (qui a perdu une case significative) mais il ne faut pas oublier de décrémenter le nombre de fils du père de $feuille(nb)$.

```
let calculer_Prufer racine fils freres =
  let n = vect_length fils
  and peres = calculer_peres racine fils freres
  and arites = calculer_arites fils freres in
  let Prufer = make_vect (n-1) (-1)
  and feuilles = make_vect n (-1)
  and nb = ref 0 in
    for k=(n-1) downto 0 do
      if (fils.(k) = -1) then
        begin
          feuilles.(!nb) <- k;
          incr nb
        end
    done;
    for j=0 to (n-2) do
      Prufer.(j) <- peres.(feuilles.(!nb-1));
      decr nb;
      if (arites.(Prufer.(j)) = 1)
        then
          nb := inserer feuilles (!nb) (Prufer.(j))
        else
          arites.(Prufer.(j)) <- pred arites.(Prufer.(j))
    done;
    Prufer
;;
calculer_Prufer : int -> int vect -> int vect -> int vect = <fun>
```

26) La création des tableaux *peres* et *arites* est linéaire de même clairement que celle du tableau $feuilles$ initial.
Ensuite chacun des $n-1$ tours de boucle est soit à temps constant (si pas d'insertion) soit d'une complexité proportionnelle à la valeur actuelle de nb.
Or $nb \leqslant nb_0$ où nb_0 est le nombre de feuilles initiales.
La complexité est donc majorée par $3n+(n-1)+p\times nb_0$ où p est le nombre d'insertions égal au nombre de nœuds internes.
Or $p+nb_0 = n-1$ donc $p\times nb_0$ est maximal équivalent à $n^2/4$ lorsque le nombre de nœuds internes et le nombre de feuilles de l'arbre initial sont environ égaux.
Ainsi la complexité est en $O(n^2)$ possiblement atteinte pour arbre de hauteur 2 tel que chaque nœud interne admette un unique fils.

Examinons les deux situations extrêmes :
Dans le cas d'un arbre "linéaire", p est maximal égal à $n-1$ mais $nb_0 = 1$ et dans le cas d'un arbre de hauteur 1, p est nul.
Dans ces deux cas la complexité est donc linéaire.

Seconde partie : du codage de Prufer au coadge racine-fils-frères.

27) Par construction même du codage de Prüfer, celui ne contient que les nœuds internes et la racine et cela un nombre de fois égal au nombre de fils.

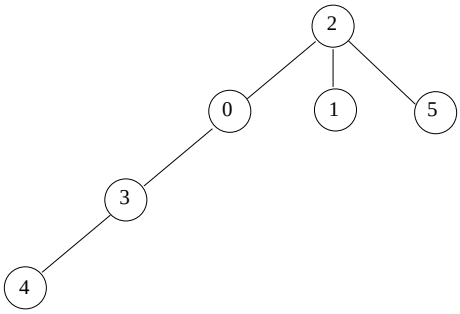
D'où la fonction suivante :

```
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)) <- succ arites.(Prufer.(i))
    done;
    arites
;;
calculer_arites_par_Prufer : int vect -> int vect = <fun>
```

28) Comme l'arbre est étiqueté consécutivement, les nœuds sont 0, 1, 2, 3, 4 et 5. On peut tout de suite dire que :

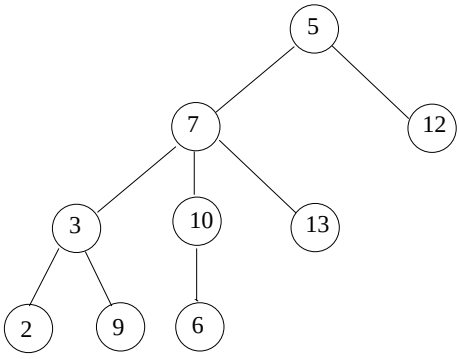
- Les feuilles sont 1, 4 et 5 (les nœuds ne figurant pas dans le codage de Prufer).
- La racine est 2 (dernier élément du codage de Prufer).
- 0 a un fils (car 0 figure une seule fois dans le codage), 2 a trois fils et 3 a un fils.

Par construction du codage de Prufer, le premier 2 du codage est le père de la feuille 1. Or 2 a trois fils. Le fils 1 étant supprimé, il reste 4 et 5 comme feuilles et donc le père de 4 est 3. Or 3 a un seul fils donc les feuilles deviennent 3 et 5. Puis le père de 3 est 0. De même on obtient ensuite que le père de 0 est 2 puis que le père de 5 est 2.



$Pr(A) = 2, 3, 0, 2, 2$

29) De la même manière, on obtient l'arbre suivant :



$Pr(A) = 3, 10, 3, 7, 7, 5, 7, 5$

30) On commence par calculer le tableau *arites* puis on construit le tableau initial $feuilles$ de longueur n avec nb cases significatives contenant les feuilles initiales triées par ordre décroissant (il suffit pour cela de parcourir le tableau *arites* en sens d'indices décroissants).

Puis on parcourt le tableau Prufer : à chaque étape :

- on connait le père de $feuille(nb-1)$ ce qui permet soit de remplir une case du tableau *peres* (qui avait bien sûr été initialisé par des -1) si elle contient -1 ou une case du tableau $freres$ (également initialisé bien sûr par des -1) sinon.
- puis on maintient à jour le tableau $feuilles$ (et nb) et le tableau *arites*.

```
let calculer_arbre Prufer fils freres =
  (* initialisation des tableux fils, freres, arites initiaux *)
  let n = vect_length fils
  and arites = calculer_arites_par_Prufer Prufer in
    for i = 0 to n-1 do fils.(i) <- -1; freres.(i) <- -1 done;
    (* création du tableau feuilles initial *)
    let feuilles = make_vect (n-1) (-1) and nb = ref 0 in
      for i=(n-1) downto 0 do
        if (arites.(i) = 0) then begin feuilles.(!nb) <- i; incr nb end
      done;
      (* construction de l'arbre *)
      for k=0 to (n-2) do
        (* gestion des tableaux fils et freres *)
        if (fils.(Prufer.(k)) = -1) then fils.(Prufer.(k)) <- feuilles.((!nb)-1)
        else
          begin
            let fr = ref fils.(Prufer.(k)) in
              while (freres.(!fr) <> -1) do fr := freres.(!fr) done;
              freres.(!fr) <- feuilles.((!nb)-1)
          end;
        (* mise à jour des tableaux feuilles et arites *)
        decr nb;
        arites.(Prufer.(k)) <- pred arites.(Prufer.(k));
        if (arites.(Prufer.(k)) = 0) then nb := inserer feuilles (!nb) Prufer.(k);
      done;
      feuilles.(0) (* ou Prufer.(n-2) *)
;;
calculer_arbre : int vect -> int vect -> int vect -> int = <fun>
```

31) L'algorithme de la question 25) montre que Pr est une application injective de $A(E)$ dans $S(E)$. En effet c'est trivialement vrai si $n = 1$. Supposons que ce soit vrai jusqu'au rang $n-1$ et soient a et b deux éléments de $A(E)$, avec $\text{card}(E) = n \geqslant 2$, ayant même codage C de Prufer. Ils ont la même plus petite feuille f car il s'agit (par construction du codage de Prufer) du plus petit entier de E ne figurant pas dans C. Alors les arbres a' et b' obtenus respectivement à partir de a et b en retirant la feuille f ont même codage de Prufer : C' obtenu à partir de C en retirant le premier élément. Par hypothèse de récurrence ils sont égaux et ainsi $a = b$.

On montre également par récurrence sur n que l'application Pr est surjective. C'est bien vrai pour $n = 1$. Supposons que ce soit vrai jusqu'au rang $(n-1)$ (avec $n \geqslant 2$) et soit C une suite de $n-1$ éléments de E. Soit c_1 le premier élément de C, C' la suite obtenue à partir de C en retirant c_1 et f le plus petit élément de E ne figurant pas dans C. Par hypothèse de récurrence il existe un arbre a' tel que $Pr(a') = C'$ dont les nœuds sont $E \setminus \{f\}$. Or $c_1 \in E \setminus \{f\}$et $f \notin C$ donc c_1 est un nœud de a'. Soit alors a l'arbre obtenu en greffant f comme fils du nœud c_1 de a'. Par construction du codage de Prufer, on a $Pr(a) = C$.

Ainsi Pr est bien une bijection de $A(E)$ sur $S(E)$ □

32) Il en découle immédaitement que $\text{card}\, A(E) = n^{n-1}$. □

Remarque 1 : Pour $n = 3$ on obtient $\text{card}(E) = 3^2 = 9$ ce qui est bien en accord avec la question 17).

Remarque 2 : La fonction `calculer_arbre` de la question 30) est la mise en œuvre de la bijection réciproque Pr^{-1} qui fournit une représentation de l'arbre dans laquelle les fils d'un même nœud sont classés par ordre croissant de gauche à droite.

Reamarque 3 : Cette fonction `calculer_arbre` ne s'applique que pour des arbres étiquetés consécutivement i.e. avec $E = \{0, 1, \ldots, n-1\}$. Considérons un ensemble E de n entiers distincts deux à deux quelconque et un codage C de Prufer relatif à E.
La bijection f strictement croissante de $\{0, 1, \ldots, n-1\}$ sur E permet clairement d'établir une bijection f_A de $A(E)$ sur l'ensemble A_n des arbres enracinés non ordonnés étiquetés consécutivement et ayant n nœuds ainsi qu'une bijection f_P de $S(E)$ sur S_n l'ensemble des suites de longueur $n-1$ dont tous les éléments sont dans $\{0, 1, \ldots, n-1\}$.
Il sufft alors de considérer $f_A^{-1}\Big(calculer\_arbre\big(f_P(C)\big)\Big)$.

FIN