École Polytechnique — concours 2001 option informatique

Corrigé par Laurent Chéno, professeur au lycée Louis-le-Grand, Paris

I Vérification de la compatibilité des flots

Question 1

Remarquons qu'une (toute) petite erreur s'est glissée dans l'énoncé, puisqu'en CAML la définition d'un type-synonyme utilise `==` et non pas `=`.

La fonction de comptage des feuilles est facile.

Programme 1 La fonction `compte_feuilles`

```
type arbre =
  | Feuille
  | Interne of arbre * arbre ;;

type flot == int vect ;;

let rec compte_feuilles = function
  | Feuille -> 1
  | Interne(a,b) -> (compte_feuilles a) + (compte_feuilles b) ;;
```

Question 2

Notons au passage qu'ici encore l'énoncé est incorrect : si la généralisation du flot f à une fonction de même nom des sous-arbres de l'arbre a entier dans $\mathbb{N}$ est clairement définie (puisque fonctionnelle), en revanche la découverte d'un nœud x vérifiant $f(x) = 0$ et provoquant donc l'arrêt de la fonction `compatible` dépend évidemment de l'ordre dans lequel nous évaluons récursivement ladite fonction. La même difficulté concerne la question 4 de cette première partie.
Ceci n'est pas formellement précisé par l'énoncé. Nous choisissons le choix le plus naturel dans *l'esprit* de l'énoncé : c'est-à-dire l'ordre postfixe qui est imposé pour l'étiquetage des feuilles.
L'écriture de la fonction `compatible` est nettement plus difficile que celle de `compte_feuilles`. Elle mélange un parcours récursif (donc naturellement traduit par un programme purement fonctionnel) avec une *consommation* d'un flot, qui relève plutôt de la programmation impérative. La solution la plus élégante pour un programmeur CAML chevronné passe probablement par l'utilisation des `exceptions`, qui ne sont pas au programme. Il n'est pas évident que les candidats se soient attendus à trouver si rapidement une telle difficulté.

Nous proposons ici deux versions.
Une version qui utilise les exceptions, et une version encore fonctionnelle mais qui ne les utilise pas.

Programme 2 La fonction `compatible`, dans sa version *savante*

```
(* Version "savante" de la fonction compatible.

   La fonction next : unit -> int permet de consommer les éléments du flot f, c'est-à-dire
   qu'à chaque appel elle renvoie l'élément suivant du flot.
   La fonction etiquette calcule les valeurs du flot sur chaque nœud de l'arbre, dans l'ordre postfixe
   imposé par l'énoncé, mais elle se charge en outre de la verification :
   si un nœud prend la valeur 0, l'exception Incompatible est déclenchée,
   ce qui assure que les calculs s'arrêtent aussitôt.
   NB: cette vérification est inutile aux feuilles, puisque le flot ne contient pas d'entier nul. *)

exception Incompatible ;;

let compatible a f =
  let indice = ref 0 in
  let next () =
    let i = !indice in
      incr indice ;
      f.(i)
  in
  let rec etiquette = function
    | Feuille -> next()
    | Interne(a,b) ->
    let i = etiquette a in
    let j = etiquette b in
    let k = (i + j) mod 4 in
      if k = 0 then raise Incompatible
      else k
  in
    try let sans_importance = etiquette a in true
    with Incompatible -> false ;;
```

Programme 3 La fonction `compatible`, dans sa version *modeste*

```
(* Version "modeste" de la fonction compatible.

   Ici, la fonction etiquette : int -> arbre -> int * int * bool
   prend en argument l'indice du prochain élément du flot à utiliser
   et l'arbre à étiqueter et rend un triplet (x,i,b)
   où x est la valeur calculée pour le nœud courant,
   où i est l'indice du prochain élément du flot à utiliser,
   et où b est faux si une incompatibilité a été décelée.    *)

let compatible a f =
  let rec etiquette i = function
    | Feuille -> (f.(i),i + 1,true)
    | Interne(a,b) ->
    let (x,j,r) = etiquette i a in
      if r then let (y,_,r') = etiquette j b in
        if r' then  let z = (x + y) mod 4 in
          (z,j,z <> 0)
        else (0,0,false)
      else (0,0,false)
  in
  let (_,_,ok) = etiquette 0 a in
    ok ;;
```

Question 3

3.a Tout nœud interne a deux fils, qui sont soit des feuilles, soit des nœuds internes. Réciproquement, tout nœud/feuille a un père sauf la racine de l'arbre, et donc si m désigne le nombre de nœuds internes, on a $2m = m + n - 1$, d'où $m = n - 1$. (On peut aussi raisonner par induction structurelle.)

3.b Le tableau suivant récapitule les différentes possibilités :

$f(a)$	couples $(f(g), f(d))$ possibles
1	(2,3), (3,2)
2	(1,1), (3,3)
3	(1,2), (2,1)

3.c Appliquant la réponse précédente, on en déduit que si $a = (g, d)$ alors

$$\begin{aligned} F(a,1) &= F(g,2)F(d,3) + F(g,3)F(d,2) \\ F(a,2) &= F(g,1)F(d,1) + F(g,3)F(d,3) \\ F(a,3) &= F(g,1)F(d,2) + F(g,2)F(d,1) \end{aligned}$$

et, bien entendu, si a est une feuille, on a : $F(a, 1) = F(a, 2) = F(a, 3) = 1$.
Ces relations permettent de déterminer par récurrence les fonctions $a \mapsto F(a, v)$ pour $v \in \{1, 2, 3\}$.
On vérifie aisément que $F(a, v) = 2^{n-1}$ où n est le nombre de feuilles de l'arbre a est solution du système, puisque si l'arbre g a p feuilles et l'arbre d en a q, l'arbre $a = (g, d)$ en possède $n = p + q$ et qu'on a bien $2^{p-1}2^{q-1} + 2^{p-1}2^{q-1} = 2^{n-1}$.
Finalement le nombre de flots compatibles avec a s'écrit $F(a, 1) + F(a, 2) + F(a, 3) = 3.2^{n-1}$.

Question 4

Ici encore, l'énoncé suppose qu'on utilise une exploration postfixe de l'arbre pour l'évaluation de la fonction `compatible`.

4.a $\nu_1(a)$ compte les flots tels que le premier calcul conduit à une somme nulle : c'est que les deux premiers éléments du flot valent $(1, 3)$, $(2, 2)$ ou $(3, 1)$. Il y a bien sûr $\nu_1(a) = 3.3^{n-2}$ tels flots. Ceci ne dépend pas de la forme de a dès qu'il n'est pas réduit à une feuille. (Si a est une feuille, bien entendu, $\nu_1(a) = 0$ car tout flot est compatible.)

4.b On suppose ici que $n \geqslant 2$, pour se donner une chance de construire un arbre à $n - 1$ feuilles.
On construit a' à partir de a en supprimant les deux premières feuilles lues consécutivement dans l'ordre postfixe **et qui ont le même père**, et en remplaçant ce père commun par une feuille. Nous noterons i et $i + 1$ les numéros (entre 1 et n) de ces feuilles de l'arbre a : ce sont les deux termes de la première somme $f_i + f_{i+1}$ calculée par la fonction `compatible` quand elle est appliquée à a.
Voici, de façon plus formelle, l'algorithme en CAML qui calcule le couple (a', i) à partir de a :

Programme 4 La fonction qui permet de calculer (a', i) à partir de a

```
let a'i_de a =
  let rec aux k = function
    | Interne(Feuille,Feuille) -> (Feuille, (k + 1), true)
    | Interne(g,d) -> let g', k', b = aux k g in
        if b then (Interne(g',d), k', b)
        else let d', k'', b = aux k' d in (Interne(g',d'), k'', b)
    | Feuille -> Feuille, (k + 1), false
  in
    match aux 0 a with (a', i, _) -> (a', i) ;;
```

La figure suivante montre un exemple d'arbre a et d'arbre a' associé. On y met en évidence aussi le flot $f' = \varphi_a(f)$ obtenu à partir de f en remplaçant les deux éléments f_i et f_{i+1} par leur somme (modulo 4).
L'arbre a' représente le schéma du calcul qu'il reste à faire à `compatible` une fois la première addition effectuée. C'est dire que si $N_+(a, f) = k \geqslant 2$ alors $N_+(a', f') = k - 1$. Il suffit pour conclure d'observer

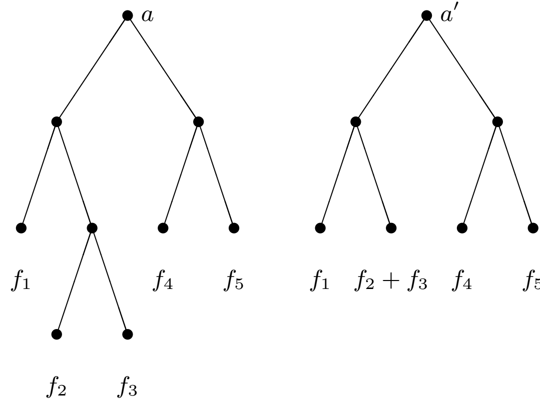

FIG. 1: un arbre a traversé par un flot f, l'arbre a' traversé par le flot $\varphi_a(f)$

qu'il y a toujours exactement deux flots qui ont la même image par φ_a. Ainsi a-t-on montré que pour $k \geqslant 2$, on dispose de $\nu_k(a) = 2\nu_{k-1}(a')$.
On se rappelle que $\nu_1(a)$ vaut toujours 3.3^{n-2} pour un arbre a possédant $n \geqslant 2$ feuilles.
On en déduit aussitôt que si a est un arbre de $n \geqslant 1$ feuilles :

$$\nu_k(a) = \begin{cases} 2^{k-1}.3.3^{n-(k-1)-2} = 3^{n-1}(\frac{2}{3})^{k-1}, & \text{si } 1 \leqslant k \leqslant n-1\,; \\ 0, & \text{sinon.} \end{cases}$$

4.c Quand $k \leqslant n-2$, il y a $\nu_k(a)$ flots f tels que $N_+(a,f) = k$. En revanche, aux $\nu_{n-1}(a)$ flots f **incompatibles** tels que $N_+(a,f) = n-1$, il faut ajouter les flots f compatibles avec a, pour lesquels on a encore $N_+(a,f) = n-1$ (puisque l'on aura fait autant d'additions qu'il y a de nœuds internes), qui sont au nombre de $F(a,1) + F(a,2) + F(a,3) = 3.2^{n-1}$.
La complexité moyenne s'écrit :

$$\begin{aligned} c_n &= \sum_f \frac{1}{3^n} N_+(a,f) = \frac{1}{3^n}\left(\sum_{k=1}^{n-1} k\nu_k(a) + 3.2^{n-1}\right) \\ &= \frac{1}{3}\sum_{k=1}^{n-1} k(2/3)^{k-1} + (2/3)^{n-1} \\ &= 3 - (n+1)(2/3)^{n-1}. \end{aligned}$$

Bien entendu, $\lim c_n = 3$.

II Triangulation des polygones

Question 5

On vérifie *à la main* que $p(3) = 0$, $p(4) = 1$ et $p(5) = 2$. Montrons que plus généralement $p(n) = n-3$ (pour $n \geqslant 3$ bien sûr).
Remarquons que dans une triangulation de P_n il existe au moins une face qui est limitée par deux côtés (consécutifs) du polygone.

Pour les sceptiques, voici une démonstration...
En effet, dans le cas contraire, à chaque côté $(i, i+1)$ (ou $(n,1)$) on associe le sommet $j(i)$ tel que $(i, i+1, j(i))$ soient les sommets de la face qui s'appuie sur le côté $(i, i+1)$, et (modulo n), on aura $j(i) \notin \{i-1, i+2\}$. Soit alors $a_1 = 1$ et $b_1 = j(a_1)$, de sorte que (a_1, a_1+1, b_1) est une face de la triangulation.
Alors $a_2 = j(b_1 - 1) \geqslant a_1 + 1$ pour éviter tout croisement et $a_2 + 1 \neq b_1 - 1$ à cause de l'hypothèse, et $b_2 = j(a_2) \leqslant b_1 - 1$ pour la même raison.
Bref, les suites (a_k) et (b_k) sont strictement monotones de monotonie contraire, et on construit ainsi deux listes infinies de sommets dans un polygone : c'est la contradiction souhaitée.

Procédons par récurrence : dans une triangulation de P_n, on supprime une face triangulaire constituée de deux côtés (consécutifs) de P_n et d'une corde, obtenant ainsi une triangulation de P_{n-1}. C'est-à-dire que $p(n) = 1 + p(n-1)$ et la récurrence s'enclenche.

Question 6

D'après l'énoncé, la tâche de `triangulation` consiste à vérifier que les cordes fournies sont bien non-sécantes et qu'il y en a bien $p(n) = n - 3$.
Or deux cordes distinctes (i, j) et (i', j') avec $1 \leqslant i < j \leqslant n$ et $1 \leqslant i' < j' \leqslant n$ sont sécantes si $i < i' < j < j'$ ou si $i' < i < j' < j$.
On en déduit le programme suivant.

Programme 5 La fonction qui permet de calculer (a', i) à partir de a

```
type segment == int * int
and segments == segment list ;;

let secantes (i,j) (i',j') =
  (i < i' && i' < j && j < j') || (i' < i && i < j' && j' < i) ;;

let non_secantes a b = not (secantes a b) ;;

(* for_all : ('a -> bool) -> 'a list -> bool
   teste si le prédicat fourni est vérifié par tous les éléments de la liste
   et s'arrête dès que le test échoue *)

let rec for_all predicat = function
  | [] -> true
  | t :: q -> (predicat t) && (for_all predicat q) ;;

(* controle : ('a -> 'a -> bool) -> 'a list -> bool
  teste si la propriété fournie est vérifiée par tous les couples (x,y)
  d'éléments distincts de la liste (x figurant avant y) *)

let rec controle propriete = function
  | [] -> true
  | [ a ] -> true
  | t :: q -> (controle propriete q) && (for_all (propriete t) q) ;;

let triangulation n sl =
  list_length sl = n - 3 && controle non_secantes sl ;;
```

La complexité de `for_all` est $O(n)$, où n est la longueur de la liste argument, donc celle de `controle` est $O(n^2)$ et celle de triangulation aussi.

Question 7

Remarquons que le dessin proposé est homéomorphe à la triangulation qu'il schématise : on a en quelque sorte *étiré* le côté $(1, 8)$, sans *rompre* aucun autre segment.

7.a Sous chaque arc du dessin proposé, on repère les deux arcs juste dessous : sous $(1, 8)$, on voit $(1, 6)$ et $(6, 8)$, sous $(1, 6)$ on voit $(1, 3)$ et $(3, 6)$, etc.
Ceci correspond à l'arbre suivant.

7.b On va associer à la triangulation t un arbre a étiqueté par les segments de la triangulation. La racine est étiquetée par le segment $(1, n)$. Si un nœud est étiqueté par (i, j) : ou bien (i, j) est un côté du polygone (autre que $(1, n)$) et ce nœud est une feuille, ou bien il s'agit d'une corde commune à deux triangles de la triangulation. Le père de ce nœud est un côté d'un de ces deux triangles, les fils de ce nœud seront les deux autres côtés de l'autre triangle.
Voici une description de l'algorithme sans doute plus conforme à l'esprit de l'énoncé : le côté $(1, n)$ de la triangulation t fait partie d'un triangle $(1, j, n)$. Si on supprime le côté $(1, n)$ on obtient un ensemble de côtés qui constituent une triangulation t_1 du polygone P_1 de sommets $(1, 2, \ldots, j-1, j)$ d'une part et une triangulation t_2 du polygone P_2 de sommets $(j, j+1, \ldots, n-1, n)$ d'autre part. Notons qu'il se peut que t_1 (ou t_2) se réduise à un seul côté.

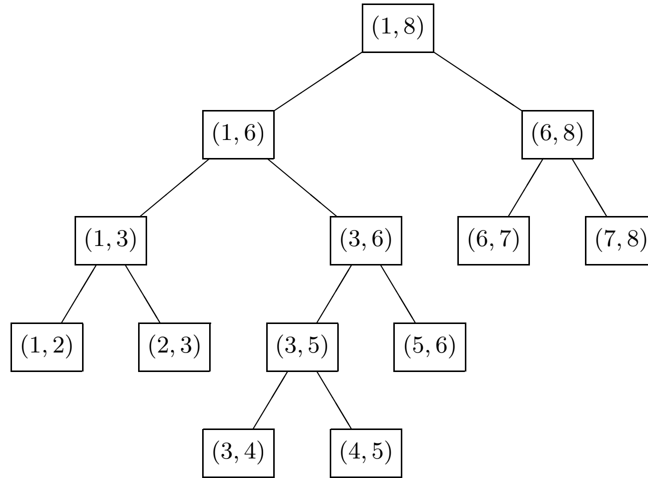

Fig. 2: l'arbre associé à la triangulation de l'énoncé

À la triangulation t on associe alors l'arbre constitué d'un nœud interne dont le fils gauche est l'arbre récursivement associé à t_1 et le fils droit l'arbre récursivement associé à t_2. Dans le cas où t_1 (ou t_2) est constitué d'un seul côté, on lui aura associé une feuille.
Remarquons pour terminer que la difficulté consiste à déterminer j. En fait j peut se calculer grâce aux relations

$$j = \max\{k < n,\ (1,k) \text{ est un côté de } t\} = \min\{k > 1,\ (k,n) \text{ est un côté de } t\}.$$

7.c On traduit l'algorithme décrit ci-dessus en Caml.

Question 8

Nous proposons ici deux versions du programme demandé.
Une première solution consiste à décorer l'arbre argument avec des couples d'entiers de sorte qu'on obtienne un arbre comme celui de la figure ci-dessus. Cela nécessite bien entendu de définir le nouveau type `arbre_decore` correspondant. Il suffit alors de lister les étiquettes de tous les nœuds de l'arbre. C'est probablement la solution la plus facile à comprendre et à imaginer.
Il est possible de ne parcourir l'arbre qu'une fois, dans une récursion postfixe, et de construire au fur et à mesure la liste des segments reconstitués. C'est plus difficile, et fait l'objet du programme suivant.
Dans les deux cas on pourra souhaiter faire un tri en ordre lexicographique du résultat (comme le fait l'énoncé sans le dire), par exemple en utilisant un tri-fusion, comme ci-dessous.

Programme 6 La fonction `triangle_arbre`

```
(* i est ici le numéro du premier sommet apparaissant dans la liste de côtés sl
   j est de même le numéro du dernier sommet
   calcul_max calcule le plus grand numéro inférieur à j d'un sommet lié à i *)

let rec calcul_max i j = function
  | [] -> 0
  | (x,y) :: q ->
      if x = i && y < j then max y (calcul_max i j q)
      else calcul_max i j q ;;

(* separe conserve des côtés qui figurent dans sl
   ceux qui connectent des sommets de numéros compris (au sens large) entre i et j *)

let rec separe i j = function
  | [] -> []
  | (x,y) :: q ->
      if x < i || y > j then separe i j q
      else (x,y) :: (separe i j q) ;;

let triangle_arbre n sl =
  let rec ta i j sl =
    if list_length sl = 1 then Feuille
    else
    let k = calcul_max i j sl
    in
    let t1 = separe i k sl
    and t2 = separe k j sl
    in
    Interne(ta i k t1, ta k j t2)
  in
    ta 1 n sl ;;
```

Programme 7 La fonction `arbre_triangle`, première version

```
(* Première version de arbre_triangle,
   en décorant les arbres par des couples d'entiers,
   au prix de deux traversées de l'arbre *)

type arbre_decore =
  | Vide
  | Noeud of segment * arbre_decore * arbre_decore ;;

let decore a =
  let indice = ref 0
  in
  let dernier = function
    | Vide -> failwith "erreur de programmation"
    | Noeud((_,d),_,_) -> d
  in
  let rec decoration x = function
    | Feuille -> Noeud((x,x+1),Vide,Vide)
    | Interne(g,d) ->
        let g' = decoration x g in
        let d' = decoration (dernier g') d in
          Noeud((x,dernier d'),g',d')
  in
    decoration 1 a ;;

let rec listage_arbre_decore = function
  | Vide -> []
  | Noeud(s,g,d) -> s :: (listage_arbre_decore g) @ (listage_arbre_decore d) ;;

let arbre_triangle a = listage_arbre_decore (decore a) ;;
```

Programme 8 La fonction `arbre_triangle`, deuxième version

```
(* Deuxième version de arbre_triangle,
   qui ne traverse l'arbre qu'une fois.
   La fonction récursive auxiliaire parcourt : segments -> arbre -> segment
   prend en arguments
   la liste des segments déjà construits,
   le sous-arbre à parcourir dans une récursion postfixe,
   et renvoie la liste des segments englobant le sous-arbre,
   le plus grand segment figurant en tête *)

let arbre_triangle a =
  let indice = ref 0
  in
  let next () = incr indice ; !indice
  in
  let rec parcourt l = function
    | Feuille -> let i = next () in (i,i+1) :: l
    | Interne(g,d) ->
        let l' = parcourt l g in
        let mini = fst (hd l') in
        let l'' = parcourt l' d in
        let maxi = snd (hd l'') in
          (mini,maxi) :: l''
  in
    parcourt [] a ;;
```

Programme 9 le tri en ordre lexicographique

```
let rec tri_fusion inf l =
  let rec dedouble = function
    | [] -> [], []
    | [a] -> [a], []
    | a :: b :: q -> let l1,l2 = dedouble q in (a :: l1), (b :: l2)
  in
  let rec fusionne = function
    | ([],l) -> l
    | (l,[]) -> l
    | (a :: q, b :: r) ->
        if inf a b then a :: (fusionne (q,(b :: r)))
        else b :: (fusionne ((a :: q),r))
  in
    match l with
      | [] -> []
      | [a] -> [a]
      | _ -> let (l1,l2) = dedouble l in fusionne (tri_fusion inf l1, tri_fusion inf l2) ;;

let tri_segments = tri_fusion (fun (i,j) (i',j') -> i < i' || (i = i' && j < j')) ;;
```

III Les quatre couleurs

Question 9

9.a Se donner deux bijections $\varphi : [0 \dots c-1] \to E$ et $\varphi' : [0 \dots c'-1] \to E'$, c'est numéroter les éléments de E et de E'.
On peut numéroter les couples de $E \times E'$ dans l'ordre lexicographique : $(e_0, e'_0), \dots, (e_0, e'_{c'-1}), (e_1, e'_0), \dots)$.
Cela revient à construire :

$$\psi : \begin{pmatrix} [0 \dots cc'-1] & \to & E \times E' \\ k & \mapsto & (\varphi(k/c'), \varphi'(k \bmod c')) \end{pmatrix}$$

9.b De façon tout à fait évidente :

$$\psi : \left(\begin{array}{ccc} [0 \dots c+c'-1] & \to & E \cup E' \\ k & \mapsto & \begin{cases} \varphi(k), & \text{si } k < c\,; \\ \varphi'(k-c), & \text{si } k \geqslant c\,; \end{cases} \end{array} \right.$$

définit une génération de $E \cup E'$ si les deux ensembles sont disjoints.

9.c Si E et E' sont de même cardinal, on peut les ranger en alternance :

$$\psi : \left(\begin{array}{ccc} [0 \dots c+c'-1] & \to & E \cup E' \\ k & \mapsto & \begin{cases} \varphi(k/2), & \text{si } k \text{ est pair}\,; \\ \varphi'((k-1)/2), & \text{si } k \text{ est impair}\,; \end{cases} \end{array} \right.$$

répond à la question.

Question 10

10.a Il s'agit de l'habituelle récurrence sur les nombres de Catalan, qui figure dans le cours :

$$N_a(n) = \begin{cases} 1, & \text{si } n = 1\,; \\ \displaystyle\sum_{k=1}^{n-1} N_a(k) N_a(n-k), & \text{si } n \geqslant 2. \end{cases}$$

Elle traduit l'égalité (pour $n \geqslant 2$) :

$$A_n = \bigcup_{1 \leqslant k \leqslant n-1} \bigcup_{\substack{g \in A_k \\ d \in A_{n-k}}} \texttt{Interne}(g, d),$$

où les unions écrites sont disjointes.
On en déduit la fonction de calcul suivante.

Programme 10 la fonction `calcule_na`

```
let calcule_na n =
  na.(1) <- 1 ;
  for p = 2 to n do
    na.(p) <- 0;
    for k = 1 to p - 1 do
      na.(p) <- na.(p) + na.(k) * na.(p - k - 1)
    done
  done ;;
```

10.b Réécrivant $A_n = \bigcup_{1 \leqslant k \leqslant n-1} \{\texttt{Interne}(g, d), g \in A_k, d \in A_{n-k}\}$, nous obtenons, en nous guidant sur les réponses à la question 9, le programme de génération suivant.

Programme 11 la fonction `int_arbre`

```
let produit_arbres (phi,c) (phi',c') =
  ((function k -> Interne (phi(k / c'), phi'(k mod c'))), c * c') ;;

let union_disjointe (phi,c)  (phi',c') =
  ((function k -> if k < c then phi k else phi' (k - c)), c + c') ;;

(* Dans l'appel int_arbre n k
   il n'est pas fait de vérification que k < na.(n), conformément à l'énoncé.
   Cette fonction suppose qu'on a calculé autant d'éléments du tableau na que nécessaire. *)

let rec int_arbre n =
  if n = 1 then function _ -> Feuille
  else if n = 2 then function _ -> Interne(Feuille,Feuille)
  else let total = ref ((function k -> Feuille),0) in
    for j = 1 to n - 1 do
      let phi = produit_arbres ((int_arbre j),na.(j)) ((int_arbre (n -j)),na.(n-j)) in
        total := union_disjointe !total phi
    done ;
    fst !total ;;
```

Question 11

On reprend ici les résultats de la question 3.
Notant $\mathcal{F}(a, v)$ l'ensemble des flots compatibles avec l'arbre a et qui prennent la valeur v à la racine, on dispose des relations suivantes (où $\bullet$ dénote la concaténation des flots) :

$$\mathcal{F}(Feuille, v) = (v), \qquad \begin{cases} \mathcal{F}(\texttt{Interne}(g,d),1) & = & \mathcal{F}(g,2) \bullet \mathcal{F}(d,3) \bigcup \mathcal{F}(g,3) \bullet \mathcal{F}(d,2) \\ \mathcal{F}(\texttt{Interne}(g,d),2) & = & \mathcal{F}(g,1) \bullet \mathcal{F}(d,1) \bigcup \mathcal{F}(g,3) \bullet \mathcal{F}(d,3) \\ \mathcal{F}(\texttt{Interne}(g,d),3) & = & \mathcal{F}(g,1) \bullet \mathcal{F}(d,2) \bigcup \mathcal{F}(g,2) \bullet \mathcal{F}(d,1) \end{cases}$$

les unions écrites étant disjointes et concernant toujours des ensembles de même cardinal.
On en déduit le programme de génération de flots qui suit.

Programme 12 la fonction `int_arbre`

```
let produit_flots (phi,c) (phi',c') =
  ((function k -> (phi (k / c')) @ (phi' (k mod c'))), c * c') ;;

let union_equipotents (phi,c) (phi',c') =
  ((function k -> if k mod 2 = 0 then phi (k / 2) else phi' (k / 2)), c + c') ;;

let rec int_flot_rec a v = match a with
  | Feuille -> ((function _ -> [ v ]), 1)
  | Interne(g,d) -> match v with
      | 1 ->
          union_equipotents
          (produit_flots (int_flot_rec g 2) (int_flot_rec d 3))
          (produit_flots (int_flot_rec g 3) (int_flot_rec d 2))
      | 2 ->
          union_equipotents
          (produit_flots (int_flot_rec g 1) (int_flot_rec d 1))
          (produit_flots (int_flot_rec g 3) (int_flot_rec d 3))
      | 3 ->
          union_equipotents
          (produit_flots (int_flot_rec g 1) (int_flot_rec d 2))
          (produit_flots (int_flot_rec g 2) (int_flot_rec d 1))
      | _ -> failwith "valeur de v interdite" ;;

let int_flot_list n a v k = (fst ( int_flot_rec a v)) k ;;

let int_flot n a v k = vect_of_list (int_flot_list n a v k) ;;
```

Question 12

12.a Il suffit d'utiliser ce qui précède. La terminaison est garantie par l'utilisation d'une boucle `for`.

Programme 13 la fonction `trouve_compatible`

```
let int_3_flots n a k = int_flot_list n a (k mod 3 + 1) (k / 3) ;;

(* un décalage binaire à gauche (logical shift left) est équivalent à une multiplication par 2,
   donc 3 lsl (n - 1) vaut bien 3 * 2 ^ (n-1) *)

let nb_flots n = 3 lsl (n - 1) ;;

exception Trouvé of int vect ;;

let trouve_compatible n a b =
  let fs = int_3_flots n a in
    try
      for k = 0 to nb_flots n - 1 do
        let f = vect_of_list (fs k) in if compatible b f then raise (Trouvé f)
      done ;
      failwith "Pas de flot compatible"
    with (Trouvé f) -> f ;;
```

Question 13

La seule nouveauté est la création d'un arbre aléatoire : on utilise pour cela le générateur d'arbres construit plus haut.

Une toute petite erreur s'est encore glissée dans l'énoncé : il faut écrire `random__int` et non pas `random_int` en CamlLight.

Programme 14 la fonction `quatre_couleurs`

```
let quatre_couleurs_utile n =
  let a = int_arbre n (random__int na.(n))
  and b = int_arbre n (random__int na.(n))
  in
    (trouve_compatible n a b),(a,b) ;;

let quatre_couleurs n = fst (quatre_couleurs_utile n) ;;
```