Corrigé : composition d'informatique (X 2001)

Partie I. Validation de la compatibilité des flots

Question 1.

```
let rec compte_feuille = function
  | Feuille        -> 1
  | Interne (g, d) -> compte_feuille g + compte_feuille d ;;
```

Question 2. Pour calculer la fonction associée au flot f pour l'arbre a, il est nécessaire de parcourir ce dernier par ordre postfixe. On utilise une référence pour indiquer l'indice de la case à utiliser lorsqu'on rencontre une feuille.

La fonction **aux** appliquée à un nœud x retourne la valeur de $f(x)$; une exception est déclenchée lorsque $f(x) = 0$, ce qui interrompt le parcours de l'arbre.

```
let compatible a f =
  let k = ref 0 in
  let rec aux = function
    | Feuille        -> let x = f.(!k) in incr k ; x
    | Interne (g, d) -> let x1 = aux g in
                        let x2 = aux d in
                        let x = (x1 + x2) mod 4 in
                        if x = 0 then raise Exit else x
  in try let _ = aux a in true
     with Exit -> false ;;
```

Notez que la syntaxe « `let x = (aux g + aux d) mod 4` » ne permet pas de garantir un parcours postfixe de la gauche vers la droite, car *l'ordre d'évaluation des arguments n'est pas spécifié en* Caml. Par exemple, les implémentations actuelles d'OCaml évaluent les arguments de la gauche vers la droite alors que l'implémentation de Caml light que j'utilise évalue de la droite vers la gauche. Ce n'est pas étonnant : dans un langage fonctionnel l'ordre d'évaluation importe peu, mais ici la fonction écrite présente un aspect impératif avec l'usage d'une référence.

Question 3.

a) Soit x le nombre de nœuds internes de a. Chacun d'eux a deux fils (nœuds ou feuilles) et chaque nœud ou feuille, *à l'exception de la racine*, possède un père donc $x + n - 1 = 2x$, soit $x = n - 1$.

b) Il suffit d'énumérer chacune des possibilités :

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

c) Si a est réduit à une feuille, $\mathrm{F}(a,v) = 1$. D'après la question précédente, si $a = (g,d)$ n'est pas réduit à une feuille alors :

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

et il n'est dès lors pas compliqué de prouver par induction structurelle que pour tout $v \in \{1,2,3\}$, $\mathrm{F}(a,v) = 2^{n-1}$ (si g a p feuilles et d a q feuilles, alors $n = p + q$ et $2^{p-1} \times 2^{q-1} + 2^{q-1} \times 2^{p-1} = 2^{n-1}$).
Le nombre de flots compatibles avec a est donc égal à $\mathrm{F}(a,1) + \mathrm{F}(a,2) + \mathrm{F}(a,3) = 3 \times 2^{n-1}$.

Question 4. Observons que le nombre d'additions modulo 4 effectuées est majoré par le nombre de nœuds internes, c'est-à-dire par $n-1$, et que si f est compatible avec a alors $\mathrm{N}_+(a,f) = n-1$.

a) Soit f un flot incompatible avec a tel que $\mathrm{N}_+(a,f) = 1$. La première addition réalisée doit donner 0 ; il s'agit donc de l'addition de deux feuilles dont les valeurs sont $(1,3)$, $(2,2)$ ou $(3,1)$, feuilles dont la position dans a est parfaitement déterminée : ce sont les fils du premier nœud interne a être traité dans le parcours postfixe. Leurs positions dans le flot est donc elle aussi parfaitement déterminée ; quant aux $n-2$ autres valeurs du flots, elles n'importent pas.
Le nombre total de tels flots est donc égal à $3 \times 3^{n-2}$, soit $\nu_1(a) = 3^{n-1}$. Cette valeur ne dépend pas de a.

b) Soit $k \geqslant 2$ et f un flot incompatible avec a tel que $\mathrm{N}_+(a,f) = k$. Puisque $k \geqslant 2$, la première addition, correspondant au premier nœud interne traité, n'a pas donné 0. Supprimons ses deux fils (qui sont des feuilles) de a pour créer un arbre $a' \in \mathrm{A}_{n-1}$, et notons $(i, i+1)$ les indices de ces deux feuilles dans le flot f.
Considérons maintenant l'application qui au flot f (de taille n) associe le flot f' (de taille $n-1$) obtenu en remplaçant (f_i, f_{i+1}) par $f_i + f_{i+1}$. Alors $\mathrm{N}_+(a', f') = k-1$.
Le tableau ci-dessous montre que chaque flot f' possède deux antécédents :

f_i	f_{i+1}	$f_i + f_{i+1}$
2	3	1
3	2	1
1	1	2
3	3	2
1	2	3
2	1	3

donc $\nu_k(a) = 2\nu_{k-1}(a')$. Compte tenu de la question précédente, il n'est dès lors plus compliqué de prouver par récurrence sur k que $\nu_k(a) = 2^{k-1} \times 3^{n-k}$.

c) D'après la question 3, le nombre de flots compatibles avec a est égal à $3 \times 2^{n-1}$, et chacun nécessite $n-1$ additions. On a donc :

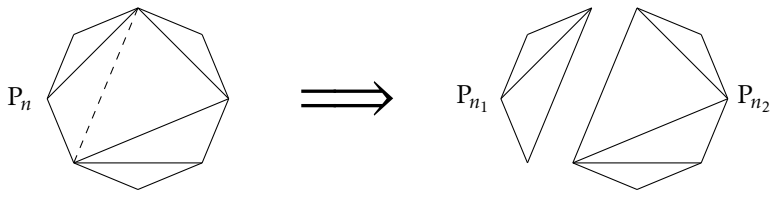
$$\sum_f \frac{1}{3^n}\mathrm{N}_+(a,f) = \sum_{k=1}^{n-1} \frac{1}{3^n} k.\nu_k(a) + \frac{1}{3^n}(n-1).(3 \times 2^{n-1}) = \frac{1}{3}\sum_{k=1}^{n-1} k\Big(\frac{2}{3}\Big)^{k-1} + (n-1)\Big(\frac{2}{3}\Big)^{n-1}.$$

D'après la formule admise, $\displaystyle\sum_f \frac{1}{3^n}\mathrm{N}_+(a,f) = 3\Big(1 - \frac{n+2}{3}\Big(\frac{2}{3}\Big)^{n-1}\Big) + (n-1)\Big(\frac{2}{3}\Big)^{n-1} = 3 - 3\Big(\frac{2}{3}\Big)^{n-1}$ et bien entendu :

$$\lim_{n\to+\infty} \sum_f \frac{1}{3^n}\mathrm{N}_+(a,f) = 3.$$

Partie II. Triangulation des polygones

Question 5. Considérons une triangulation de P_n (avec $n \geqslant 4$) ainsi qu'une de ses cordes (il en possède au moins une). Celle-ci sépare P_n en deux polygones triangulés P_{n_1} et P_{n_2} avec $n_1 \geqslant 3$, $n_2 \geqslant 3$ et $n_1 + n_2 = n+2$.



Si, comme le suggère l'énoncé, le nombre de cordes $p(n)$ ne dépend que de n, nous aurons $p(n) = p(n_1) + p(n_2) + 1$.
Il reste à deviner la formule ; or il est immédiat de constater que $p(3) = 0$, $p(4) = 1$, $p(5) = 2$, ce qui suggère que $p(n) = n-3$.
Il reste alors à prouver par récurrence forte que : « *si* $n \geqslant 3$, *toute triangulation de* P_n *utilise* $n-3$ *cordes* », ce qui ne présente plus de difficulté.

Question 6. Compte tenu du résultat admis, il reste à vérifier deux points : la liste de segments est de longueur $n-3$ et aucun des segments n'en rencontre un autre. Il est facile de constater que les segments (i,j) et (k,ℓ) ne se rencontrent pas si et seulement si $(k \in]i,j[$ et $\ell \in]i,j[)$ ou $(k \notin]i,j[$ et $\ell \notin]i,j[)$, ce qui permet de définir la fonction :

```
let ne_se_coupent_pas (i, j) (k, l) =
  let est_dans (i, j) k = i < k && k < j in
  (est_dans (i, j) k && est_dans (i, j) l)
  || (not est_dans (i, j) k && not est_dans (i, j) l) ;;
```

On définit alors :

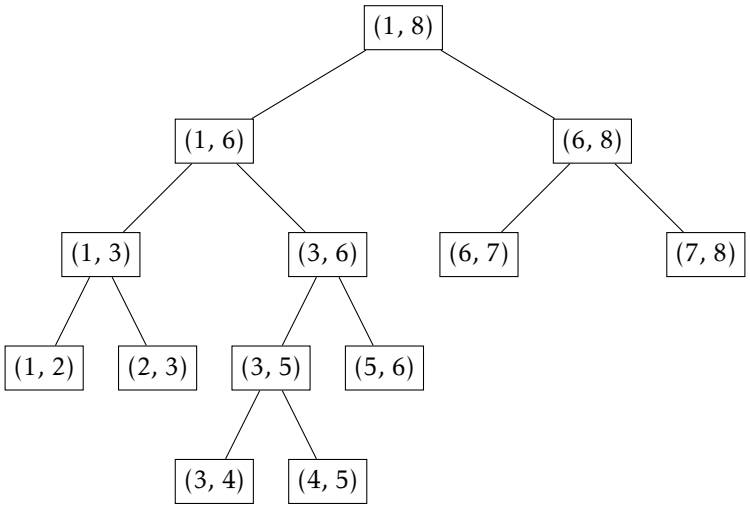
```
let rec triangulation n = function
  | []   -> n = 3
  | t::q -> for_all (ne_se_coupent_pas t) q && (triangulation (n-1) q) ;;
```

La fonctionnelle **`for_all`** est de type *('a -> bool) -> 'a list -> bool* :

`for_all f [a1; ...; an]` est équivalent à **`f a1 && ... && f an`**.

Question 7.

a) Chaque segment de la triangulation est un nœud de l'arbre, et on convient que chaque nœud est le père des nœuds qu'il chapeaute. On obtient l'arbre suivant :



qui est un arbre binaire à 7 feuilles, donc un élément de A_7.

b) Dans cette question, P_n désigne un polygone convexe à n côtés ($n \geqslant 2$) dont les sommets sont des entiers consécutifs *ne débutant pas nécessairement par 1*. Si $n = 2$, $P_2 = (i, i+1)$ se réduit à un segment.

À tout polygone triangulé P_n de ce type on associe un arbre a de A_{n-1} en suivant l'algorithme :

(i) Si $n = 2$, à $P_2 = (i, i+1)$ on associe la feuille $(i, i+1)$.

(ii) Si $n > 2$ et $P_n = (i, i+1, \ldots, j)$, soit $k \in [\![i+1, j-1]\!]$ tel que le triangle (i,k,j) fasse partie de la triangulation. On associe alors à P_n l'arbre dont :

- la racine est le segment (i,j) ;
- le fils gauche est l'arbre associé au polygone triangulé $P_{n_1} = (i, i+1, \ldots, k)$;
- le fils droit est l'arbre associé au polygone triangulé $P_{n_2} = (k, k+1, \ldots, j)$.

Il est facile de prouver par récurrence sur n que l'arbre ainsi construit est élément de A_{n-1} car $(n_1 - 1) + (n_2 - 1) = n - 1$.

c) Il reste à mettre en œuvre cet algorithme. Le polygone $P_n = (i, i+1, \ldots, j)$ sera décrit par le couple (i,j), et l'entier k sera caractérisé par le fait qu'il s'agit de l'unique entier pour lequel les segments (i,k) et (k,j) sont présents dans la triangulation. On le calcule à l'aide de la fonction :

```
let rec trouver_k i j segs =
  let rec aux = function
    | k when mem (i, k) segs && mem (k, j) segs -> k
    | k -> aux (k+1)
  in aux i ;;
```

et la fonction principale s'écrit :

```
let triangle_arbre n segs =
  let rec aux i = function
    | j when j = i + 1 -> Feuille
    | j -> let k = trouver_k i j segs in
           Interne (aux i k, aux k j)
  in aux 1 n ;;
```

`aux i j` calcule l'arbre associé au polygone de sommets $i, i+1, \ldots, j$.

Question 8. Pour la fonction réciproque, on propose :

```
let arbre_triangle a =
  let rec aux i = function
    | Feuille        -> i+1, [(i, i+1)]
    | Interne (g, d) -> let k, s1 = aux i g in
                        let j, s2 = aux k d in
                        j, (i, j)::s1@s2
  in snd (aux 1 a) ;;
```

La fonction `aux i a` retourne un couple formé de l'indice du dernier sommet et de la triangulation associée à l'arbre a lorsqu'on numérote les sommets à partir de l'indice i. Si on veut éviter le coût de la concaténation, on peut transporter la liste des segments déjà ajoutés à l'aide d'un accumulateur, mais comme on ne s'intéresse pas au coût de la fonction j'ai privilégié la simplicité de l'écriture.

Partie III. Les quatre couleurs

Question 9. Dans cette question on pose $c = |\mathrm{E}|$ et $c' = |\mathrm{E}'|$.

a) Une génération de E consiste à ordonner les éléments de cet ensemble. Dès lors, il est naturel d'ordonner les éléments de $\mathrm{E} \times \mathrm{E}'$ suivant l'ordre lexicographique induit par ϕ et ϕ' en posant :

$$\psi(n) = (\phi(q), \phi'(r)) \qquad \text{avec} \quad n = qc' + r \text{ et } 0 \leqslant r < c'$$

b) On ordonne $\mathrm{E} \cup \mathrm{E}'$ en commençant par les éléments de E, puis par ceux de E' :

$$\psi(n) = \begin{cases} \phi(n) & \text{si } n \leqslant c \\ \phi'(n-c) & \text{si } n > c \end{cases}$$

c) Lorsque $c = c'$, on entrelace les éléments de E et de E' en posant :

$$\psi(n) = \begin{cases} \phi(n/2) & \text{si } n \text{ est pair} \\ \phi'((n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

Question 10.

a) Si $n = 1$, A_1 se réduit à une feuille donc $\mathrm{N}_a(1) = 1$.
Si $n > 1$, tout arbre a de A_n est constitué d'un couple d'arbres (g, d) où $g \in \mathrm{A}_k$ et $d \in \mathrm{A}_{n-k}$, avec $1 \leqslant k \leqslant n-1$. D'où la relation :

$$\mathrm{N}_a(n) = \sum_{k=1}^{n-1} \mathrm{N}_a(k)\mathrm{N}_a(n-k).$$

Remarque. On reconnait la suite de Catalan dont une forme close est : $\mathrm{N}_a(n) = \dfrac{1}{n+1}\binom{2n}{n}$.

On calcule les termes de cette suite par mémoïsation :

```
let rec calcule_na = function
  | 1 -> na.(1) <- 1
  | n -> calcule_na (n-1) ;
         for k = 1 to n-1 do na.(n) <- na.(n) + na.(k) * na.(n-k) done ;;
```

b) Reprenons le raisonnement ci-dessus : si $A_{n,k}$ désigne les arbres (g,d) de A_n pour lesquels $g \in A_k$, nous avons $A_n = \bigcup_{k=1}^{n-1} A_{n,k}$, cette union étant disjointe. Nous allons donc utiliser la génération de l'union disjointe décrite au 9.b.
Par ailleurs, $A_{n,k}$ est en bijection évidente avec $A_k \times A_{n-k}$; nous allons utiliser la génération du produit cartésien décrite à la question 9.a pour obtenir une génération de $A_{n,k}$.
La fonction suivante calcule l'unique entier i vérifiant : $\sum_{j=1}^{i-1} N_a(j)N_a(n-j) \leqslant k < \sum_{j=1}^{i} N_a(j)N_a(n-j)$ et retourne le couple (i,k') avec $k' = k - \sum_{j=1}^{i-1} N_a(j)N_a(n-j)$:

```
let union_disjointe n k =
  let rec aux acc = function
    | i when acc + na.(i) * na.(n-i) > k -> i, k - acc
    | i -> aux (acc + na.(i) * na.(n-i)) (i+1)
  in aux 0 1 ;;
```

Il s'agit ensuite de générer le couple $(g,d) \in A_i \times A_{n-i}$ de rang k' :

```
let rec int_arbre n k = match n with
  | 1 -> Feuille
  | n -> let i, kprime = union_disjointe n k in
         let g = int_arbre i (kprime mod na.(i))
         and d = int_arbre (n-i) (kprime / na.(i)) in
         Interne (g, d) ;;
```

Question 11. Posons $a = (g,d)$. D'après la question 3, l'ensemble $F_{a,v}$ des flots compatibles avec a et tel que $f(a) = v$ est en bijection avec l'union disjointe de $F_{g,v_1} \times F_{d,v_2}$ et de $F_{g,v_1'} \times F_{d,v_2'}$, où les couples (v_1,v_2) et (v_1',v_2') s'expriment en fonction de v dans le tableau de la question 3b. Puisque cette union concerne deux ensembles de même cardinal, nous pouvons générer cet ensemble par la technique décrite à la question 9c.
On commence par une fonction qui, en fonction de v et de k, retourne les valeurs de $f(g)$ et $f(d)$:

```
let entrelac k = function
  | 1 when k mod 2 = 0 -> (2, 3)
  | 1                  -> (3, 2)
  | 2 when k mod 2 = 0 -> (1, 1)
  | 2                  -> (3, 3)
  | 3 when k mod 2 = 0 -> (1, 2)
  | _                  -> (2, 1) ;;
```

La fonction principale s'écrit alors :

```
let rec int_flot n a v k =
  let f = make_vect n 0 in
  let i = ref 0 in
  let rec aux v k = function
    | Feuille        -> f.(!i) <- v ; incr i
    | Interne (g, d) -> let (v1, v2) = entrelac k v in
                        aux v1 (k/2) g ; aux v2 (k/2) d
  in aux v k a ; f ;;
```

Question 12.

a) Nous savons (question 3c) qu'il y a 3.2^{n-1} flots compatibles avec un arbre a à n feuilles, que l'on peut générer à l'aide de la fonction précédente (en faisant varier v entre 1 et 3) ; la compatibilité avec b de chacun des ces flots peut être testée avec la fonction **`compatible`** de la question 2. Tout ceci donne :

```
let trouve_compatible n a b =
  let fav = 1 lsl (n-1) in
  let rec aux = function
    | (3, k) when k = fav -> failwith "trouve_compatible"
    | (v, k) when k = fav -> aux (v+1, 0)
    | (v, k) ->let f = int_flot n a v k in
               if compatible b f then f else aux (v, k+1)
  in aux (1, 0) ;;
```

Remarque. L'opérateur **`lsl`** (*logical shift left*) décale l'écriture binaire d'un entier vers la droite ; ainsi, **`1 lsl p`** retourne l'entier $(1\underbrace{00\cdots00}_{p})_2$, c'est-à-dire 2^p.

b) Il reste à combiner la fonction **`int_arbre`** avec la fonction précédente :

```
let quatre_couleurs 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 ;;
```