X 2001 - Corrigé

```
type arbre = Feuille | Interne of arbre * arbre ;;
type flot == int vect ;;
```

I. Vérification de la compatibilité des flots

Question 1

```
let rec compte_feuilles a =
  match a with
    Feuille -> 1
  | Interne (g, d) -> compte_feuilles g + compte_feuilles d ;;
```

Question 2 La difficulté, pour la fonction `compatible`, et de penser à avoir un entier (appelé ici `index`) pour savoir quel est le numéro de la feuille rencontrée.

Pour arrêter efficacement le parcours de l'arbre en cas d'incompatibilité, on peut utiliser une exception.

```
exception Pas_compatible ;;

let compatible arbre flot =
  let rec compat_aux arb index =
    match arb with
      Feuille -> (flot.(index), index+1)
    | Interne (a1, a2) ->
        let (v1, i1) = compat_aux a1 index in
        let (v2, i2) = compat_aux a2 i1 in
        let v = (v1 + v2) mod 4 in
        if v = 0 then raise Pas_compatible ;
        (v, i2)
  in
  try
    let _ = compat_aux arbre 0 in true
  with Pas_compatible -> false ;;
```

Question 3

a) On procède par induction structurelle. Soit a un arbre de A_n. S'il s'agit d'une feuille, alors $n = 1$ et a possède $0 = n - 1$ nœuds internes. Sinon, si n_g (resp. n_d) désigne le nombre de feuilles du sous-arbre gauche a_g (resp. du sous-arbre droit a_d), alors a_g possède $n_g - 1$ nœuds internes et a_d en possède $n_d - 1$. Ainsi, a possède $1 + (n_g - 1) + (n_d - 1)$ nœuds internes, soit au total $n_g + n_d - 1 = n - 1$.

Remarque On peut procéder sans récurrence. Si il y a i nœuds internes, il y a en tout $i + n$ nœuds. Mais si l'on regarde les nœuds issus de chaque nœud interne, on obtient tous les nœuds sauf la racine. On a ainsi $1 + 2i = n + i$.

b) Pour une valeur $f(a)$ fixée, il faut que $f(g)$ et $f(d)$ soient tous deux différents de $f(a)$. Les deux valeurs restantes sont possibles, et on a :
- Si $f(a) = 1$, on a $(f(g), f(d)) \in \{(2;3);(3;2)\}$;
- Si $f(a) = 2$, on a $(f(g), f(d)) \in \{(1;1);(3;3)\}$;
- Si $f(a) = 3$, on a $(f(g), f(d)) \in \{(1;2);(2;1)\}$;

c) Montrons que pour tout entier $v \in \{1;2;3\}$, on a $F(a, v) = 2^{n-1}$ où n est le nombre de feuilles de a. Si $n = 1$, on a $F(a, v) = 1 = 2^{1-1}$ puisque l'arbre est constitué d'une unique feuille dont la valeur pour le flot considéré est forcément v. Sinon, notons $a = (f, g)$. D'après la question précédent, il y a deux façons d'affecter des valeurs de flot compatible à g et à d. Comme le nombre de flots compatibles ne dépend pas de la valeur v considérée, on en déduit, en l'écrivant « à la Caml », que :

$$F(a, v) = 2 \times F(g, \_) \times F(d, \_) = 2 \times 2^{n_g - 1} \times 2^{n_d - 1} = 2^{1 + (n_g - 1) + (n_d - 1)} = 2^{n-1}$$

On en déduit que le nombre de flots compatibles est égal à $3 \times 2^{n-1}$ pour un arbre ayant n feuilles.

Question 4

a) La fonction `compatible` effectue un parcours préfixe (ou postfixe) de l'arbre, et la première addition effectuée par la fonction est celle correspondant au premier nœud interne rencontré ayant deux feuilles comme fils. Ainsi, pour découvrir qu'un flot est incompatible au bout d'une addition, il faut que les feuilles correspondant à ce nœud interne aient des valeurs pour le flot dont la somme vaut 4 (puisque l'on fait les additions modulo 4 et que les valeurs des feuilles sont comprises entre 1 et 3). Les seules possibilités sont $1+3$, $2+2$ et $3+1$. Comme les $n-2$ autres feuilles peuvent avoir n'importe quelle valeur dans $\{1;2;3\}$, on en déduit que :

$$v_1(a) = 3^{n-1}$$

En particulier, cette valeur ne dépend pas de l'arbre considéré.

b) D'après la discussion précédente, si $k \geq 2$, cela signifie que le flot restreint au premier nœud interne ayant deux feuilles comme fils est compatible. Soit donc a' l'arbre obtenu à partir de a en remplaçant ce nœud interne par une feuille. À chaque flot f' sur a' tel que $N_+(a', f') \geq 1$ correspondent deux flots f sur a tels que $N_+(a, f) \geq 2$, comme l'a montré la question précédente. On a donc bien $v_k(a) = 2v_{k-1}(a')$.

Ainsi, on déduit par récurrence qu'il existe un arbre $a^{(k-1)}$ ayant $n-(k-1)$ feuilles, et tel que :

$$v_k(a) = 2^{k-1} v_1(a^{(k-1)}) = 2^{k-1} \times 3^{n-(k-1)-1} = 2^{k-1} \times 3^{n-k}$$

Remarque On peut vérifier en passant que l'on obtient bien ainsi le bon nombre de flots. Si un arbre a a n feuilles, il a 3^n flots. Or, on a dénombré $3 \times 2^{n-1}$ flots compatibles (3 valeurs sont possibles pour la racine et il y a 2^{n-1} façons de compléter le flot) et le nombre de flots incompatibles est :

$$\sum_{k=1}^{n-1} v_k(a) = \sum_{k=1}^{n-1} 2^{k-1} \times 3^{n-k} = \frac{3^n}{2} \sum_{k=1}^{n-1} \left(\frac{2}{3}\right)^k = 3^{n-1} \times \frac{1-\left(\frac{2}{3}\right)^{n-1}}{1-\frac{2}{3}} = 3^n - 3 \times 2^{n-1}$$

On a donc correctement dénombré les flots de a.

c) Les $3 \times 2^{n-1}$ flots compatibles nécessitent tous $n-1$ additions pour vérifier leur compatibilité effective. Pour les flots incompatibles, pour $k \in [\![1; n-1]\!]$, il y en a $v_k(a)$ qui nécessitent k additions pour montrer leur incompatibilité. On a donc :

$$\sum_f N_+(a,f) = 3 \times 2^{n-1} \times (n-1) + \sum_{k=1}^{n-1} 2^{k-1} \times 3^{n-k} \times k = 3 \times 2^{n-1} \times (n-1) + 3^{n-1} \times \sum_{k=1}^{n-1} k \left(\frac{2}{3}\right)^{k-1}$$

D'après l'indication, on a :

$$3^{n-1} \sum_{k=1}^{n-1} k \left(\frac{2}{3}\right)^{k-1} = 3^{n-1} \times \frac{1-\left(\frac{2}{3}\right)^{n-1} \times \left(n-\frac{2}{3}(n-1)\right)}{\left(1-\frac{2}{3}\right)^2} = 3^{n+1} - 3 \times 2^{n-1} \times n - 3 \times 2^n$$

On en déduit la complexité moyenne :

$$\frac{3 \times 2^{n-1} \times (n-1) + 3^{n+1} - 3 \times 2^{n-1} \times n - 3 \times 2^n}{3^n} = 3 - 3 \times \left(\frac{2}{3}\right)^{n-1}$$

En particulier, la limite de la complexité moyenne lorsque n tend vers ∞ est égale à 3.

Remarque Il est aisé de redémontrer l'indication :

$$\sum_{k=1}^{n} k\alpha^{k-1} = \frac{1-\alpha^n(n+1-\alpha n)}{(1-\alpha)^2}$$

En effet, on a : $\sum_{k=0}^{n} \alpha^k = \frac{1-\alpha^{n+1}}{1-\alpha}$. Si l'on voit ces deux expressions comme des fonctions de α et si on les dérive, on obtient :

$$\sum_{k=0}^{n} k\alpha^{k-1} = \frac{-(n+1)\alpha^n \times (1-\alpha) - (1-\alpha^{n+1}) \times -1}{(1-\alpha)^2} = \frac{1-\alpha^n(n+1-\alpha n)}{(1-\alpha)^2}$$

II. Triangulation de polygones

Question 5 Montrons par récurrence qu'une triangulation de P_n a $p(n) = n-3$ cordes. Pour $n = 3$, c'est clairement le cas.
Sinon, supposons le résultat vrai pour un entier $n \geq 3$, et considérons une triangulation T de P_{n+1}. Soit a une arête de P_{n+1}. Cette arête appartient à une unique face f, et deux cas sont possibles pour cette face :

– Si cette face est composée de a, d'une autre arête et d'une corde c, on considère le polygone P_n et la triangulation T' obtenus en enlevant les deux arêtes de f (la corde c devenant une arête du nouveau polygone). La triangulation T' a une corde en moins par rapport à T (la corde c étant devenue une arête), on a donc par récurrence :

$$p(n+1) = p(n) + 1 = (n+1) - 3$$

– Si cette face est en plus composée de deux autres arêtes, on obtient en supprimant a deux nouveaux polygones de la forme P_k et P_{n+2-k} avec $k \geq 3$ et $n+2-k \geq 3$ (autrement dit avec $3 \leq k \leq n-1$) ainsi que deux triangulations correspondantes T' et T''. Comme on a $k \leq n$ et $n+2-k \leq n$, on peut appliquer la récurrence, et donc T' et T'' ont respectivement $k-3$ et $n-1-k$ cordes. La triangulation de départ T a donc $2+(k-3)+(n-1-k) = n-2 = (n+1)-3$ cordes. Ainsi, $p(n+1) = (n+1)-3$.

Remarque Cette solution est une version très classique. Une autre façon de procéder est, lors de la récurrence, de choisir une corde de la triangulation. Elle sépare notre polygone triangulé à $n+1$ côtés en deux polygones triangulés d'au moins n côtes, ce qui permet d'utiliser la récurrence.

Remarque Une troisième méthode, élégante, sans récurrence et qui préfigure la question **7** b), consiste à remarquer que si l'on a n côtés, c cordes et t triangles, alors comme chaque côté est dans 1 triangle et chaque corde est dans 2 triangles, on a $3t = n+2c$. D'autre part, en partant d'un côté de la triangulation, on regarde le triangle correspondant, et chaque corde côté de ce triangle donne accès à un nouveau triangle. En continuant ainsi, on va rencontrer chaque corde qui correspondra à un triangle, et on aura rencontré chaque triangle. On a ainsi $t = c+1$. On déduit le résultat de ces deux égalités.

Question 6 Il est clair que vérifier qu'une liste de $p(n)$ cordes est bien une triangulation revient à vérifier qu'il n'y a pas de cordes sécantes dans la liste fournie. Or, deux cordes (a,b) et (c,d) sont sécantes si, et seulement si l'une des valeurs c ou d est dans l'intervalle $]a;b[$ et l'autre n'est pas dans $[a;b]$. On en déduit la fonction `triangulation` :

```
let rec triangulation n l =
  match l with
    [] -> n = 3 (* P(n) = n-3 *)
  | (a, b) :: l' ->
    let rec triang_aux l =
```

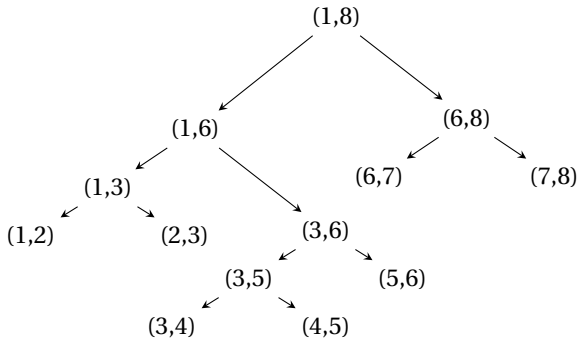
```
  match l with
    [] -> true
  | (c, d) :: l' ->
    if (c < a && a < d && d < b) || (a < c && c < b && b < d)
    then false else triang_aux l'
in (triangulation (n-1) l') && (triang_aux l') ;;
```

Dans la fonction précédente, on si la liste est non vide, on commence par appeler `triangulation` avec la queue de la liste. Cela permet de vérifier en premier que la longueur de la liste est correcte, sans effectuer un parcours de liste inutile.

La complexité de `triangulation` est bien en $O(n^2)$, puisque pour chaque élément de la liste, on effectue un test en temps constant sur chacun des éléments qui se trouve à sa droite.

Question 7

a) On peut associer à l'exemple fourni l'arbre suivant :



b) Partant d'un côté du polygone, ce côté appartient à une unique face et on peut lui associer deux segments, les deux autres côtés de la face. Si un de ces segments est une corde, on peut lui associer une nouvelle face (une corde faisant partie de deux faces). On peut ainsi répéter le processus. Il est clair que tous les segments seront atteints de cette façon, et qu'ils ne le seront qu'une fois, puisqu'un segment n'appartient au plus qu'à deux faces.

c) On en déduit la fonction suivante :

```
let triangle_arbre n t =
  let rec aux debut fin n =
    if n = fin
    then Feuille (* n'arrive que si fin = debut + 1 *)
    else
      if (mem (debut, n) t) && (mem (n, fin) t)
      then Interne (aux debut n (debut+1), aux n fin (n+1))
      else aux debut fin (n+1)
  in aux 1 n 2 ;;
```

Question 8 La fonction inverse s'exécute en remarquant que les numéros des sommets s'obtiennent en comptant le nombre de feuilles.

```
let arbre_triangle arbre =
  let rec aux arbre debut l =
    match arbre with
      Feuille ->  ((debut, debut + 1) :: l, debut + 1)
    | Interne (g, d) ->
        let (l1, debut1) = aux g debut l in
        let (l2, debut2) = aux d debut1 l1 in
        ((debut, debut2) :: l2, debut2)
  in
  let (segments, _) = aux arbre 1 [] in
  segments ;;
```

III. Les quatre couleurs

Question 9

a) Si l'on note n et n' les cardinaux respectifs de E et de E', une génération de $E \times E'$ est réalisée par la fonction ψ définie sur $[0, n\,n'-1]$ par :

$$\forall\,(k,k') \in [0,n-1]\times[0,n'-1],\ \psi(k+n\,k') = (\varphi(k),\varphi(k'))$$

Autrement dit, on a $\psi(k) = \left(\varphi(k \mod n), \varphi'\left(\left\lfloor \frac{k}{n} \right\rfloor\right)\right)$.

b) Avec les notations précédentes, une génération de $E \cup E'$ est donnée par la fonction définie sur $[0, n+n'-1]$ par :

$$\begin{aligned} k < n &\mapsto \varphi(k) \\ k \geq n &\mapsto \varphi'(k-n) \end{aligned}$$

c) Si les ensembles E et E' ont même cardinal, on peut considérer la fonction suivante :

$$\begin{aligned} 2k &\mapsto \varphi(k) \\ 2k+1 &\mapsto \varphi'(k) \end{aligned}$$

Question 10

a) Il existe un unique arbre avec 1 feuille. Par contre, pour $n \geq 2$, un arbre comportant n feuilles a ses deux sous-arbres comportant respectivement k et $n-k$ feuilles pour $1 \leq k \leq n-1$. On en déduit :

$$N_a(1) = 1 \qquad \forall\, n \geq 2,\ N_a(n) = \sum_{k=1}^{n-1} N_a(k) N_a(n-k)$$

On en déduit directement la fonction :

```
let rec calcule_na n =
  if n = 1 then na.(1) <- 1
  else begin
    calcule_na (n - 1) ;
    let val = ref 0 in
    for k = 1 to n - 1 do
      val := !val + na.(k) * na.(n - k)
    done ;
    na.(n) <- !val
  end ;;
```

Remarque Un œil avisé aura reconnu la suite des nombres de Catalan. Ainsi, le nombre d'arbres ayant n feuilles est $\frac{1}{n+1}\binom{2n}{n}$.

b) Pour construire une génération de A_n, nous allons utiliser à outrance le tableau `na` que l'on suppose rempli.

```
let rec int_arbre n k =
  if n = 1
  then Feuille
  else
    (* la fonction aux détermine le nombre
       de feuilles de chaque sous-arbre *)
    let rec aux i k =
      if k >= na.(i) * na.(n - i)
      then aux (i + 1) (k - na.(i) * na.(n - i))
      else (i, k)
    in
    let (i, k') = aux 1 k in
    (* on a 0 <= k' < na.(i) * na.(n - i) *)
    let g = int_arbre i (k' mod na.(i))
    and d = int_arbre (n-i) (k' / na.(i)) in
    Interne (g, d) ;;
```

Question 11 On rappelle que $F(a, v) = 2^{n-1}$. Ainsi, on peut voir l'entier k utilisé comme « numéro » du flot comme une suite de $n-1$ booléens.

```
let int_flot n a v k =
  let flot = make_vect (n + 1) 0
  and k_ref = ref k
  and pos = ref 1 in
  let rec aux a v = match a with
      Feuille -> flot.(!pos) <- v ; pos := !pos + 1
    | Interne (g, d) -> begin
        (* on lit un bit d'information *)
        let bit = !k_ref mod 2 in
        k_ref := !k_ref / 2 ;
        (* on énumère les cas de la question I. 3 c) *)
        let (vg, vd) = match (v, bit) with
            (1, 0) -> (2, 3)
          | (1, 1) -> (3, 2)
          | (2, 0) -> (1, 1)
          | (2, 1) -> (3, 3)
          | (3, 0) -> (1, 2)
          | (3, 1) -> (2, 1)
          | _ -> failwith "int_flot match"
        in
        aux g vg ; aux d vd
      end
  in
  aux a v ; flot ;;
```

Question 12

Remarque Dans cette dernière question, on se propose de vérifier expérimentalement qu'étant donné deux arbres avec le même nombre de feuilles, il existe un flot compatible avec les deux. Pour cela, on dire deux arbres au hasard dans A_n, en utilisant la génération précédente, puis on parcours les flots correspondants.

La vérification expérimentale correspond au fait que l'on trouve toujours un flot compatible.

a) On se donne la liberté de modifier le type de la fonction, en renvoyant un résultat de type `flot option`.

```
exception Flot of flot ;;

let trouve_compatible n a b =
  let max := ref 0 in
  for k = 1 to n do
    max := 2 * !ref
```

```
    done ;
    (* on peut, sinon, faire : max := 2 lsl n ; puisque
       multiplier par 2 revient à décaler vers la gauche. *)
    try
      for i = 0 to !max - 1 do
        for j = 1 to 3 do
          let flot = int_flot n a (1 + (k mod 3)) (k / 3) in
          if compatible b flot then raise Flot flot
        done
      done ;
    None
    with Flot flot -> Some flot ;;
```

b)
```
let quatre_couleurs n =
    let a1 = random__int na.(n)
    and a2 = random__int na.(n) in
    trouve_compatible n (int_arbre n a1) (int_arbre n a2) ;;
```