COMPOSITION D'INFORMATIQUE (Option) n°1

Durée : 2h

⋆ ⋆ ⋆

Exercice 1

1. On utilise une référence qui contient la valeur du maximum des i premières cases du tableau, qu'on compare à $i+1$-ème case.

```
let minimum t =
   let x = ref t.(0) in
   for i = 1 to Array.length t - 1 do
      if t.(i) < !x then x := t.(i)
   done;
   !x;;
```

2. On crée une chaîne de caractère de taille $j-i$ qu'on remplit avec une boucle `for` avec les bons caractères.

```
let sous_chaine s i j =
   let s' = String.make (j - i) 'a' in
   for k = i to j - 1 do
      s'.[k - i] <- s.[k]
   done;
   s';;
```

Exercice 2

1. Cette fonction est de signature `'a list -> int -> 'a list * 'a list`. Elle partitionne une liste $[x_1, x_2, \ldots, x_m]$ en $[x_1, \ldots, x_n], [x_{n+1}, \ldots, x_m]$.
2. Si la liste contient 0 ou 1 élément, on renvoie une erreur. Si elle en contient deux, on renvoie le premier. Sinon, l'avant-dernier de la liste est l'avant-dernier de sa queue.

```
let rec avant_dernier = function
   | [] -> failwith "Liste vide"
   | [_] -> failwith "Liste trop courte"
   | [x ; _] -> x
   | _ :: q -> avant_dernier q;
```

Problème

1. On écrit une simple fonction prenant en considération que le nombre de feuilles de (g, d) est égal au nombre de feuilles de g plus celui de d.

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

2. (a) Par récurrence :
   - Si l'arbre est une feuille, alors $f_a = 1$ et $n_a = 1$. C'est vérifié.
   - Sinon, si on suppose que le résultat est vrai pour g et d, et que $a = (g, d)$, alors :
     * $f_a = f_g + f_d$
     * $n_a = 1 + n_g + n_d = 1 + f_g - 1 + f_d - 1 = f_a - 1$

   (b) On donne le résultat sous forme de tableau :

$f(a)$	$(f(g), f(d))$
1	$\{(2,3),(3,2)\}$
2	$\{(1,1),(3,3)\}$
3	$\{(1,2),(2,1)\}$

(c) - On remarque facilement que si a est une feuille, alors $F(a,1) = F(a,2) = F(a,3) = 1 = 2^0$.

- Grâce à la question précédente, si a n'est pas une feuille on a :

$$F(a,1) = F(g,2)F(d,3) + F(g,3)F(d,2)$$
$$F(a,2) = F(g,1)F(d,1) + F(g,3)F(d,3)$$
$$F(a,3) = F(g,1)F(d,2) + F(g,2)F(d,1)$$

Si on suppose que le résultat est vrai pour g et d, alors $F(a,1) = 2^{k-1}2^{m-1} \times 2$ où $k \in [\![1, n-1]\!]$ est le nombre de feuilles de g et $m = n - k$. On en déduit que $F(a,1) = 2^{n-1}$. On raisonne de même pour $F(a,2)$ et $F(a,3)$.

On en déduit finalement que le nombre de flots compatibles avec a est $F(a) = F(a,1) + F(a,2) + F(a,3) = 3 \times 2^{n-1}$.

3. On commence à appeler la fonction sur le sous-arbre gauche pour calculer $f(g)$. La référence sera alors incrémentée du nombre de feuilles dans l'arbre gauche. On peut alors faire un appel sur le sous-arbre droit pour calculer $f(d)$. On peut calculer $f(a)$, et si cette valeur est 0, on met le booléen à `false` (cela signifie que le flot n'est pas compatible).

```
let compatible a f =
   let num_feuille = ref (-1) in
   let b = ref true in
   let rec image_par_f = function
      | Feuille -> incr num_feuille; f.(!num_feuille)
      | Interne(g, d) -> let fg = image_par_f g in
                         let fd = image_par_f d in
                         let fa = (fg + fd) mod 4 in
                         if fa = 0 then b := false; fa
   in let fa = image_par_f f a in
   !b;;
```

4. On écrit ici une fonction auxiliaire qui prend en argument un flot f, un arbre a et un booléen b. Comme il n'est pas possible d'accéder à un élément donné, il faut enlever les éléments de la liste avant de pouvoir calculer le reste des images. Le rôle de cette fonction est donc de calculer les images des différents nœuds et de remplacer les valeurs correspondantes dans la liste par l'image du parent des nœuds remplacés. Elle renvoie de plus un booléen indiquant s'il y a bien compatibilité (il joue le rôle de la référence précédente). Dans l'exemple de la figure, on a, si $a = (g, d)$:

- `aux [3;3;1;2;2] g true` renvoie `[2;1;2;2], true`, puis `aux [1;2;2] d true` renvoie `[1], true`, et finalement `aux [3;3;1;2;2] a true` renverra bien `[3], true`.
- `aux [2;3;3;2;3] g true` renvoie `[1;3;2;3], true`, puis `aux [3;2;3] d true` renvoie `[0], false`, et finalement `aux [2;3;3;2;3] a true` renverra bien `[1], false`.

On traite les différents cas qui se présentent :

- Deux cas de base : l'arbre à une feuille et l'arbre à deux feuilles. Ils se traitent de manière évidente.
- Un premier cas d'appel récursif : si l'arbre gauche est une feuille. On lance l'appel sur la queue de la liste et l'arbre droit, puis on se ramène à l'arbre à deux feuilles.
- Un deuxième cas d'appel récursif : dans le cas général, on lance d'abord un appel sur l'arbre gauche, puis on se ramène au cas précédent.

Le filtrage n'est pas exhaustif, mais cela n'empêchera pas la fonction de fonctionner. On aurait pu fusionner les 3 derniers cas, mais cela alourdit l'écriture et n'aide pas à mieux comprendre la fonction.

```
let compatible f a =
   let rec aux f a b = match f, a with
      | [x], Feuille -> [x], x <> 0 && b
      | x :: y :: q, Interne(Feuille, Feuille) -> let z = (x + y) mod 4 in
                                                  z :: q, z <> 0 && b
      | x :: q, Interne(Feuille, d) -> let fd, bd = aux q d b in
                                       aux (x :: fd) (Interne(Feuille, Feuille)) bd
      | _, Interne(g, d) -> let fg, bg = aux f g b in
                            aux fg (Interne(Feuille, d)) bg in
   let fa, b = aux f a true in
   b;;
```

5. On pose $C_n = \text{Card}(A_n)$ le nombre d'arbres à n feuilles.
   (a) Soit $n \in \mathbb{N}^*$. Alors pour construire un arbre à n feuilles, il faut choisir un entier $k \in [\![1; n-1]\!]$, un fils gauche à k feuilles et un fils droit à $n-k$ feuilles. On en déduit la formule :
   $$C_n = \sum_{k=1}^{n-1} C_k C_{n-k}$$
   (b) On crée un tableau qu'on met à jour en utilisant la formule précédente.
   ```
   let catalan n =
      let c = Array.make n 1 in
      for i = 2 to n - 1 do
         let v = ref 0 in
         for k = 1 to i - 1 do
            v := !v + c.(k) * c.(i - k)
         done;
         c.(i) <- !v
      done;
      c;;
   ```
   (c) Les multiplications ont lieu à la 6e ligne de la fonction. Il faut donc compter le nombre de passage dans cette boucle, qui vaut : $\displaystyle\sum_{i=2}^{n-1}(i-1) = \sum_{i=1}^{n-2} i = \frac{(n-2)(n-1)}{2}$.
6. Cette fonction a besoin d'une numérotation des arbres de A_n, pour associer à un numéro un arbre unique. L'idée de la fonction est de trouver, en fonction du numéro m de l'arbre cherché, les informations suivantes :
   - La taille k du sous-arbre gauche (dont on peut déduire celle du sous-arbre droit).
   - Le numéro m_g du sous-arbre gauche dans A_k.
   - Le numéro m_d du sous-arbre droit dans A_{n-k}.

   Dans un premier temps, pour trouver k, on partitionne l'ensemble $[\![0; C_n - 1]\!]$ par les ensembles $E_k = [\![S_k; S_{k+1} - 1]\!]$, pour $k \in [\![1; n-1]\!]$, où on pose $S_k = \displaystyle\sum_{i=1}^{k-1} C_i C_{k-i}$. Ainsi, l'entier m n'appartient qu'à un seul de ces E_k, et c'est cet ensemble qui donne l'entier k cherché (en effet, se retrouver dans l'ensemble E_k signifie qu'on a déjà parcouru tous les arbres de A_n avec $1, 2, \ldots, k-1$ feuilles dans le sous-arbre gauche). On écrit pour cela la fonction suivante, dans laquelle la référence `s` stocke la valeur S_k :
   ```
   let trouver_k n m =
      let k = ref 1 and s = ref 0 in
      while !k < n - 1 && !s < m do
         s := !s + c.(!k) * c.(n - !k);
         incr k
      done;
      !k;;
   ```
   Ensuite, on sait que $m_g \in [\![0; C_k - 1]\!]$ et $m_d \in [\![0; C_{n-k} - 1]\!]$ et que ces nombres doivent être en relation avec m. On a donc besoin d'une fonction bijective permettant de passer de m à m_g, m_d en

connaissant k. Or, la division euclidienne de m par C_k fournit exactement ces valeurs. On pose donc $m_g = m \mod C_k$ et $m_d = \left\lfloor \frac{m}{C_k} \right\rfloor$. On peut alors lancer des appels récursifs.

Il reste bien sûr à gérer les cas de base (pour que la fonction puisse s'arrêter) : lorsque $n = 1$, l'arbre à construire est une feuille.

```
let rec genere_arbre n m = match n with
      | 1 -> Feuille
      | _ -> let k = trouver_k n m in
             let mg = m mod c.(k) and md = m / c.(k) in
             Interne(genere_arbre k mg, genere_arbre (n - k) md);;
```

⋆ ⋆ ⋆