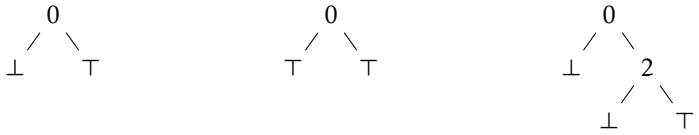
Corrigé : arbres combinatoires (X 2012)

Partie I. Arbres combinatoires

Question 1. L'arbre $(2 \to \bot, \top)$ représente l'ensemble $\{\{2\}\}$; l'arbre $(1 \to (2 \to \bot, \top), \top)$ représente donc l'ensemble $\{\{2\},\{1\}\}$ et l'arbre $(1 \to \top, (2 \to \bot, \top))$ l'ensemble $\{\emptyset, \{1,2\}\}$. Ainsi, l'arbre de l'exemple (1) représente l'ensemble $\{\{2\},\{1\},\{0\},\{0,1,2\}\}$.

Question 2. Les trois arbres combinatoires associés respectivement aux ensembles $\{\{0\}\}$, $\{\emptyset,\{0\}\}$ et $\{\{0,2\}\}$ sont représentés ci-dessous :



Question 3. Montrons par induction structurelle que tout arbre combinatoire A différent de $\bot$ contient au moins une feuille étiquetée par $\top$.

- Si l'arbre A se restreint à une feuille, celle-ci ne qu'être étiquetée que par $\top$.
- Si l'arbre A est de la forme $i \to A_1, A_2$ d'après la condition de suppression on a $A_2 \neq \bot$ donc par hypothèse d'induction A_2 (et donc A) contient une feuille étiquetée par $\top$.

Question 4. Notons u_n le nombre d'arbres combinatoires étiquetés par un ensemble E de cardinal n.
On a $u_0 = 2$ car seuls deux arbres peuvent être étiquetés par $\emptyset$: $\top$ et $\bot$.
Considérons maintenant un arbre combinatoire A étiqueté par E. S'il contient un nœud étiqueté par 0, il est nécessairement de la forme $(0 \to A_1, A_2)$ où A_1 et A_2 sont des arbres combinatoires étiquetés par $[\![1, n-1]\!]$ et $A_2 \neq \bot$. Il y a donc $u_{n-1}(u_{n-1}-1)$ arbres de ce type.
Si A ne contient par de nœud étiqueté par 0, A est un arbre combinatoire étiqueté par $[\![1, n-1]\!]$; il y a u_{n-1} arbres de ce type.
On en déduit que $u_n = u_{n-1}(u_{n-1}-1) + u_{n-1} = u_{n-1}^2$.
Les relations $u_0 = 2$ et $u_n = u_{n-1}^2$ permettent alors de prouver sans peine que $u_n = 2^{2^n}$.

Remarque. Ce résultat laisse penser que l'application S réalise une bijection entre l'ensemble des arbres combinatoires et l'ensemble $\mathscr{P}(\mathscr{P}(E))$; on peut le prouver en montrant par exemple que S est surjective.
On considère un ensemble de parties P de E et on prouve par récurrence sur l'entier $n_P = \sum_{U \in P} \operatorname{card} U$ qu'il existe un arbre combinatoire A tel que $S(A) = P$.

- Si $n_P = 0$ alors $P = \emptyset$ et dans ce cas $P = S(\bot)$ ou $P = \{\emptyset\}$ et dans ce cas $P = S(\top)$.
- Si $n_P \geqslant 1$ on suppose le résultat acquis aux rangs inférieurs, on considère le plus petit élément i de E qui appartienne à une des parties de E contenues dans P et on définit :

$$P_1 = \left\{U \in P \,\middle|\, i \notin U\right\} \qquad \text{et} \qquad P_2 = \left\{U \setminus \{i\} \,\middle|\, U \in P \text{ et } i \in U\right\}$$

Par hypothèse de récurrence il existe deux arbres combinatoires A_1 et A_2 tels que $P_1 = S(A_1)$ et $P_2 = S(A_2)$. Posons $A = (i \to A_1, A_2)$. A est bien un arbre combinatoire car il vérifie bien les propriétés d'ordre ($j \in P_1 \cup P_2 \Rightarrow i < j$) et de suppression ($A_2 \neq \bot$ car $P_2 \neq \emptyset$), et $S(A) = P$.

Partie II. Fonctions élémentaires sur les arbres combinatoires

Question 5. On choisit de descendre le long de la branche droite de l'arbre.

```
let rec un_elt = function
  | Zero            -> failwith "un_elt"
  | Un              -> []
  | Comb (i, a1, a2) -> i::(un_elt a2) ;;
```

Question 6.

```
let rec singleton = function
  | []   -> Un
  | i::q -> Comb (i, Zero, singleton q) ;;
```

Question 7.

```
let rec appartient s a = match s, a with
  | [], Un                          -> true
  | [], Comb (_, a1, _)             -> appartient [] a1
  | i::q, Comb (j, _, a2) when i = j -> appartient q a2
  | i::q, Comb (j, a1, _) when i < j -> appartient s a1
  | _                               -> false ;;
```

Question 8.

```
let rec cardinal = function
  | Zero            -> 0
  | Un              -> 1
  | Comb (_, a1, a2) -> cardinal a1 + cardinal a2 ;;
```

Partie III. Principe de mémorisation

Question 9. Les sous-arbres de l'arbre combinatoire A de l'exemple (1) sont :

$$\bot \qquad \top \qquad 2 \to \bot, \top \qquad 1 \to (2 \to \bot, \top), \top \qquad 1 \to \top, (2 \to \bot, \top) \quad \text{et} \quad A.$$

Il est donc de taille 6.

Question 10. Il s'agit du principe classique de mémoïsation : on associe à la fonction à mémoïser un dictionnaire stockant les valeurs des cardinaux des arbres déjà calculés, et on ne procède à un appel récursif que si l'arbre n'est pas déjà présent dans celui-ci.

```
let cardinal a =
  let t = cree1() in
  let rec aux = function
    | Zero                    -> 0
    | Un                      -> 1
    | a when present1 (t, a)  -> trouve1 (t, a)
    | Comb (i, a1, a2) as a   -> let c = aux a1 + aux a2 in ajoute1 (t, a, c) ; c
  in aux a ;;
```

Le nombre d'additions et d'appels à la fonction **`ajoute1`** effectués par cette fonction est égal au nombre de sous-arbres distincts de $\bot$ et $\top$ de l'arbre A, donc à $T(A) - 2$.
Le nombre d'appels aux fonctions **`present1`** et **`trouve1`** est majoré par le nombre de nœuds et de feuilles de l'arbre A, lui-même majoré par $2T(A)$ (chaque nœud possède deux fils).
Puisque les fonctions **`present1`**, **`trouve1`** et **`ajoute1`** sont de coûts constants on peut en conclure que la complexité tant temporelle que spatiale est un $O(T(A))$.

Question 11. La fonction qui suit repose sur les relations suivantes :

- si $A_1 = \bot$ ou $A_2 = \bot$ alors $A = \bot$;
- si $A_1 = \top$ et $A_2 = \top$ alors $A = \top$;
- si $A_1 = \top$ et $A_2 = j \to C_1, C_2$ alors A représente l'intersection de $\top$ et C_1 ;
- si $A_1 = i \to B_1, B_2$ et $A_2 = \top$ alors A représente l'intersection de B_1 et $\top$;
- si $A_1 = i \to B_1, B_2$ et $A_2 = j \to C_1, C_2$ avec $i < j$ alors A représente l'intersection de B_1 et A_2 ;
- si $A_1 = i \to B_1, B_2$ et $A_2 = j \to C_1, C_2$ avec $i > j$ alors A représente l'intersection de A_1 et C_1 ;
- si $A_1 = i \to B_1, B_2$ et $A_2 = i \to C_1, C_2$ on calcule l'intersection D_1 de B_1 et C_1 et l'intersection D_2 de B_2 et C_2 : si $D_2 = \bot$ alors $A = D_1$ sinon $A = i \to D_1, D_2$.

On utilise une mémoïsation des résultats pour limiter les calculs.

```
let inter a1 a2 =
  let t = cree2() in
  let rec aux a1 a2 = match (a1, a2) with
    | Zero, _                                  -> Zero
    | _, Zero                                  -> Zero
    | Un, Un                                   -> Un
    | _ when present2 (t, (a1, a2))            -> trouve2 (t, (a1, a2))
    | Un, Comb (_, c1, _)                      -> let u = aux Un c1 in
                                                  ajoute2 (t, (Un, a2), u) ; u
    | Comb (_, b1, _), Un                      -> let u = aux b1 Un in
                                                  ajoute2 (t, (a1, Un), u) ; u
    | Comb (i, b1, _), Comb (j, _, _) when i < j  -> let u = aux b1 a2 in
                                                  ajoute2 (t, (a1, a2), u) ; u
    | Comb (i, _, _), Comb (j, c1, _) when i > j  -> let u = aux a1 c1 in
                                                  ajoute2 (t, (a1, a2), u) ; u
    | Comb (i, b1, b2), Comb (_, c1, c2)       -> let d1 = aux b1 c1 and d2 = aux b2 c2 in
                                                  let u = if d2 = Zero then d1
                                                          else Comb (i, d1, d2) in
                                                  ajoute2 (t, (a1, a2), u) ; u
  in aux a1 a2 ;;
```

Question 12. Notons $A1 \cap A_2$ l'arbre combinatoire qui représente l'intersection des arbres A_1 et A_2 et considérons l'application :

$$\varphi : \left(\begin{array}{ccc} \mathcal{U}(A_1)\times\mathcal{U}(A_2) & \longrightarrow & \mathcal{U}(A_1\cap A_2) \\ (B_1,B_2) & \longmapsto & B_1\cap B_2 \end{array} \right)$$

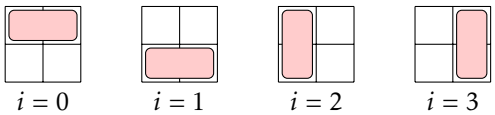
Notons déjà qu'il s'agit bien d'une application : $S(B_1\cap B_2) = S(B_1)\cap S(B_2) \subset S(A_1)\cap S(A_2) = S(A_1\cap A_2)$ donc $B_1\cap B_2$ est bien un sous-arbre de $A_1\cap A_2$.
Montrons maintenant que cette application est surjective : Si B est un sous-arbre de $A_1\cap A_2$ alors B est un sous-arbre de A_1 et de A_2 donc $(B,B)\in\mathcal{U}(A_1)\times\mathcal{U}(A_2)$ et $\varphi(B,B) = B$.
De ceci il résulte que $\operatorname{card}\mathcal{U}(A_1\cap A_2) \leqslant \operatorname{card}\mathcal{U}(A_1)\times\mathcal{U}(A_2)$, à savoir :

$$T(\texttt{inter}(A_1,A_2)) \leqslant T(A_1)\times T(A_2).$$

Partie IV. Application au dénombrement

Question 13. Si le domino est placé horizontalement, il y a (p-1) emplacements possibles sur chacune des lignes donc $p(p-1)$ en tout. Le raisonnement est analogue si le domino est placé verticalement, donc il y a en tout $2p(p-1)$ possibilités.

Pour comprendre de quoi il retourne, il peut être utile d'examiner la situation avec $p = 2$. On a ici $n = 2p(p-1) = 4$ possibilités de placer un domino, possibilités que l'on numérote *arbitrairement* avec un entier $i \in [\![0,3]\!]$:



0	1
2	3

numérotation j des cases

Le booléen m_{ij} indique si la case j est recouverte dans le placement i, ce qui donne la matrice : $\begin{pmatrix} v & v & f & f \\ f & f & v & v \\ v & f & v & f \\ f & v & f & v \end{pmatrix}$

Avec ce choix, les couvertures exactes de la matrice m sont les ensembles $\{0,1\}$ et $\{2,3\}$.

Question 14. La fonction **colonne** utilise deux fonctions auxiliaires mutuellement récursives :

- **aux1** construit un arbre combinatoire A_i vérifiant la propriété :

$$\forall s\in\mathscr{P}([\![i,n-1]\!]),\qquad s\in S(A_i) \iff \exists!k\in s \;\middle|\; m_{kj} = \texttt{true}.$$

– **aux2** construit un arbre combinatoire A'_i vérifiant la propriété :

$$\forall s \in \mathscr{P}([\![i, n-1]\!]), \qquad s \in S(A'_i) \iff \forall k \in s,\ m_{kj} = \texttt{false}.$$

```
let colonne j =
  let n = vect_length m in
  let rec aux1 = function
    | i when i = n      -> Zero
    | i when m.(i).(j) -> let a1 = aux1 (i+1) and a2 = aux2 (i+1) in Comb (i, a1, a2)
    | i                 -> let a = aux1 (i+1) in if a = Zero then Zero else Comb (i, a, a)
  and aux2 = function
    | i when i = n      -> Un
    | i when m.(i).(j) -> aux2 (i+1)
    | i                 -> let a = aux2 (i+1) in Comb (i, a, a)
  in aux1 0 ;;
```

Notons $C_1(n)$ et $C_2(n)$ les complexités respectives de ces deux fonctions auxiliaires. On dispose de la relation : $C_2(n) = C_2(n-1) + O(1)$ qui prouve que $C_2(n) = O(n)$.
Pour établir la complexité de **aux1** il faut remarquer que dans chaque colonne de m il y a au maximum 4 valeurs égales à **true** (correspondant aux différentes façons de recouvrir la case j par un domino) donc $C_1(n) \leqslant n \times O(1) + 4O(n) = O(n)$.
La fonction **colonne** est donc de complexité linéaire vis-à-vis de n.

Question 15. L'arbre A_j calculé par **colonne j** représente toutes les façons de poser des dominos sur l'échiquier (éventuellement en se chevauchant) de sorte que la case j soit couverte par un et un seul domino. L'arbre A demandé est donc l'intersection de tous ces arbres A_j.

```
let pavage () =
  let rec aux = function
    | 0 -> colonne 0
    | j -> inter (aux (j-1)) (colonne j)
  in aux (vect_length m.(0) - 1) ;;
```

On peut illustrer l'utilisation de cette fonction avec le cas $p = 2$:

```
# pavage () ;;
- : ac = Comb (0, Comb (2, Zero, Comb (3, Zero, Un)), Comb (1, Zero, Un))
```

L'arbre combinatoire retourné représente l'ensemble $\{\{0,1\},\{2,3\}\}$; ce sont bien les deux pavages possibles du carré 2×2.

Partie V. Tables de hachage

Il s'agit dans cette partie d'implémenter une table de hachage avec une résolution des collisions par chaînage. Les fonctions demandées ne présentent guère de difficulté.

Question 16.

```
let ajoute t k v = let i = hache k in t.(i) <- (k, v)::t.(i) ;;
```

Question 17.

```
let present t k =
  let rec aux = function
    | []        -> false
    | (c, _)::q -> egal c k || aux q
  in aux t.(hache k) ;;
```

Question 18.

```
let trouve t k =
  let rec aux = function
    | []                      -> failwith "trouve"
    | (c, v)::q when egal c k -> v
    | _::q                    -> aux q
  in aux t.(hache k) ;;
```

Question 19. La fonction **ajoute** est de coût constant mais les fonctions **present** et **trouve** ont un coût en $O(b)$, où b majore la longueur du plus grand des seaux. Pour qu'on puisse les considérer comme des fonctions de coûts constants il est donc nécessaire que la fonction de hachage répartisse les clés de la manière la plus uniforme possible dans les différents seaux et que le rapport entre le nombre de clés utilisées et le nombre de seaux H reste majoré par une constante.

Partie VI. Construction des arbres combinatoires

Question 20. Soient A_1 et A_2 deux arbres combinatoires tels que **egal A1 A2 = true**.
Si $A_1 = \bot$ ou $A_1 = \top$ alors $A_2 = A_1$ et on a bien **hache**(A_1) = **hache**(A_2).
Sinon on peut poser $A_1 = i \to L_1, R_1$ et $A_2 = i \to L_2, R_2$ avec **unique**(L_1) = **unique**(L_2) et **unique**(R_1) = **unique**(R_2).
On en déduit immédiatement que **hache**(A_1) = **hache**(A_2).

Question 21. Nous allons utiliser ici une référence globale **uni** qui fournit le plus petit entier non encore utilisé pour garantir l'unicité d'un arbre combinatoire, ainsi qu'une table d'association (elle aussi globale) **arbres** qui contiendra les arbres déjà construits. Pour utiliser sans les réécrire les fonctions de la partie V les clés et les valeurs seront égales et de type *ac*.
Il faut donc commencer par définir :

```
let uni = ref 2 ;;
let arbres = make_vect H [] ;;
ajoute arbres Zero Zero ;;
ajoute arbres Un Un ;;
```

La fonction **cons** consiste alors à déterminer si l'arbre qu'on cherche à construire se trouve déjà dans la table et s'il ne s'y trouve pas, à le rajouter tout en incrémentant la référence **uni** :

```
let cons i a1 a2 =
  let a = Comb(!uni, i, a1, a2) in
  if present arbres a then trouve arbres a
  else (ajoute arbres a a ; incr uni ; a) ;;
```

À titre d'illustration on peut construire l'arbre combinatoire de l'exemple 1 :

```
# cons 0 (cons 1 (cons 2 Zero Un) Un) (cons 1 Un (cons 2 Zero Un));;
- : ac = Comb (5, 0, Comb (4, 1, Comb (2, 2, Zero, Un), Un),
           Comb (3, 1, Un, Comb (2, 2, Zero, Un)))
```

Cet arbre s'est vu attribué le numéro 5 et a nécessité la création des arbres $(2 \to \bot, \top)$ (arbre numéro 2), $(1 \to \top, (2 \to \bot, \top))$ (arbre numéro 3) et $(1 \to (2 \to \bot, \top), \top)$ (arbre numéro 4).

Question 22. Nous faisons l'hypothèse que l'arbre que l'on cherche à construire a autant de chance d'aboutir dans chacun des seaux.
Si l'arbre doit être construit pour la première fois, le seau dans lequel il va être rangé doit être parcouru dans son entier, ce qui conduit à un nombre moyen de comparaisons égal à :

$$\frac{1}{19\,997}\left(0\times 6\,450 + 1\times 7\,340 + 2\times 4\,080 + 3\times 1\,617 + 4\times 400 + 5\times 96 + 6\times 11 + 7\times 3\right) \approx 1{,}13.$$

Si l'arbre est déjà présent dans la table, le seau dans lequel il se trouve va être parcouru partiellement. Si on fait l'hypothèse que sa position dans le seau est arbitraire on peut considérer qu'en moyenne la moitié des arbres du seau en question va lui être comparé, ce qui conduit à un nombre moyen de comparaisons égal à :

$$\frac{1}{19\,997}\left(0\times 6\,450 + 0{,}5\times 7\,340 + 1\times 4\,080 + 1{,}5\times 1\,617 + 2\times 400 + 2{,}5\times 96 + 3\times 11 + 3{,}5\times 3\right) \approx 0{,}56.$$