OPTION INFORMATIQUE - ENS 97 : CORRIGÉ

Première partie : épaisseur d'un arbre binaire strict

Remarques sur les arbres binaires stricts (ie localement complets) :

⋆ dans la définition de la hauteur d'un arbre binaire donnée par l'énoncé à la page 4, il ne faut pas compter la feuille. Ainsi, c'est le nombre maximum d'arêtes traversées pour aller de la racine à une feuille.

⋆ Ici on ne considère pas d'arbre vide.
La hauteur h d'un arbre binaire strict est donc toujours ≥ 1.
La racine est toujours un noeud interne.
Le nombre n_f de feuilles est égal à $n_i + 1$ où n_i désigne le nombre de noeuds internes.

Question 1

– Pour $k = 2$, il y a 7 arêtes d'épaisseur 2 dont 4 issues de noeuds de biépaisseur $(1, 2)$, donc $R_{2,1} = 4/7$ et 3 issues de noeuds de biépaisseur $(1, 1)$, donc $R_{2,2} = 3/7$.

– Pour $k = 3$, il y a 2 arêtes d'épaisseur 3 dont aucune issue de noeuds de biépaisseur $(1, 3)$, donc $R_{3,1} = 0$, une issue d'un noeud de biépaisseur $(2, 3)$, d'où $R_{3,2} = 1/2$ et une issue d'un noeud de biépaisseur $(2, 2)$, donc $R_{3,3} = 1/2$.

La matrice de ramification est donc $\begin{pmatrix} 1 & 0 & 0 \\ 4/7 & 3/7 & 0 \\ 0 & 1/2 & 1/2 \end{pmatrix}$

Question 2

La somme des éléments d'une ligne est toujours égale à **1**.

En effet, c'est évident pour la première ligne.

Pour $k \geq 2$, $\sum_{i=1}^{k} R_{k,i} = \frac{1}{N_t(k)} \left(N_t(k-1, k-1) + \sum_{i=1}^{k-1} N_t(i, k) \right) = 1$ car pour calculer $N_t(k)$, on effectue le total des noeuds de biépaiseur$(1, k), (2, k), \ldots, (k-1, k)$, ce qui donne $\sum_{i=1}^{k-1} N_t(i, k)$ et des $N_t(k-1, k-1)$ noeuds de biépaisseur $(k-1, k-1)$.

Question 3

Le fait qu'un arbre binaire strict de hauteur h soit **parfait** si et seulement si toutes les feuilles sont à une hauteur h est "évident".

Pour le "montrer" comme le demande l'énoncé, on peut rédiger une récurrence sur k .....

Dans un tel arbre, il y a 2^h feuilles et $2^h - 1$ noeuds internes.

On vérifie facilement par récurrence que, dans un arbre parfait, tous les noeuds ont une biépaisseur de la forme $(k-1, k-1)$ avec $2 \leq k \leq h$ et que l'épaisseur de cet arbre est $h + 1$.

Ainsi $N_t(k) = N_t(k-1, k-1)$: il en résulte que la matrice de ramification est la matrice unité I_{h+1}.

Question 4

On vérifie facilement par récurrence que pour un **peigne** de hauteur h (et qui est de taille $n = h$) :
- il y a h noeuds internes (avec la racine)
- le noeud interne le plus éloigné de la racine a pour biépaisseur $(1, 1)$
- les autres noeuds ont pour biépaisseur $(1, 2)$.

L'épaisseur d'un arbre-peigne est donc toujours égale à 2.

$R_{2,1} = \frac{h-1}{h}$ et $R_{2,2} = \frac{1}{h}$, d'où la matrice $\begin{pmatrix} 1 & 0 \\ (h-1)/h & 1/h \end{pmatrix}$.

Remarque : si $h = 1$, il s'agit à la fois un arbre parfait et un peigne, ce qui donne I_2 dans les deux cas.

Question 5

L'épaisseur e d'un arbre est $\leq h+1$.

Intuitivement :

- plus un arbre de hauteur h est effilé, plus son épaisseur est petite.
- lorsque les coefficients $R_{i,k}$ pour $i = 1, 2\ldots$ sont non nuls, alors l'arbre est très déséquilibré, le cas optimum $e = 2$ caractérisant les arbres dans lesquels chaque noeud interne a au moins une feuille pour fils (cf Question **6**).
- au contraire, si l'épaisseur e est "voisine" de $h+1$, alors en même temps la matrice est assez "creuse" : l'arbre est assez équilibré, donc touffu, le cas optimum $e = h+1$ caractérisant l'arbre parfait.

Question 6

n désigne la **taille** de l'arbre qui représente dans ce problème le nombre de noeuds internes.

Quand on remonte l'arbre en partant d'une feuille pour rejoindre la racine, les arêtes parcourues ont une épaisseur croissante.

L'épaisseur d'une arête qui n'aboutit pas à une feuille est au moins égale à 2.

Par conséquent $\boxed{\lambda(n) = 2}$ puique ce minimum est atteint pour un arbre peigne de hauteur $h = n$.

Cherchons le nombre μ_n d'arbres de taille n d'épaisseur 2.

$\mu_1 = 1$ et si $n \geq 1$, $\mu_n = 2\,\mu_{n-1}$ car les deux fils de la racine sont une feuille (soit placée à gauche, soit à droite) et un arbre d'épaisseur 2 de taille $n-1$ (soit placé à droite, soit à gauche).

Il y a donc $\boxed{\mu_n = 2^{n-1}}$ arbres de taille n d'épaisseur 2 : ce sont tous les arbres **dégénérés** de taille n.

Remarque : tous ces arbres de formes assez différentes ont cependant la même matrice de ramification.

Question 7

Notons e (au lieu de h comme le propose l'énoncé) l'épaisseur d'un arbre (h désignant la hauteur).

Notons N_e l'ensemble des valeurs n possibles pour la taille d'un arbre d'épaisseur $e \geq 2$ et $\alpha_e = \min N_e$.

On a déjà $N_2 = [\![1; \infty[$ et $\alpha_2 = 1$ en prenant des arbres dégénérés.

Si $e \geq 3$ et si `A` est un arbre d'épaisseur e de taille n, alors les deux fils `A1` et `A2` de la racine sont des arbres de taille respective n_1 et n_2, d'épaisseur respective e_1 et e_2 vérifiant :

$$n = 1 + n_1 + n_2 \text{ et } e = \begin{cases} e = e_1 + 1 & \text{si } e_1 = e_2 \quad \text{cas (1)} \\ e = \max\{e_1, e_2\} & \text{si } e_1 \neq n_2 \quad \text{cas (2)} \end{cases}$$

Dans le premier cas , $n \in 1 + N_{e-1} + N_{e-1}$ et dans le second : $n \in 1 + N_e + \bigcup_{k=2}^{e-1} N_k$.

Le premier cas permet d'imaginer que l'on a $\alpha_e = 1 + 2\,\alpha_{e-1}$ (suite arithmético-géométrique) et donc que $\alpha_e = 2^{e-1} - 1$ et que toutes les valeurs n supérieures à α_e sont possibles.

Conjecturons donc que $\boxed{N_e = [\![2^{e-1} - 1; \infty[}$. C'est déjà vrai pour $e = 2$.

Démontrons par récurrence (forte) sur e cette égalité (par double inclusion).

Supposons que $\forall\, k \in [\![2; e-1]\!]$, $N_k = [\![2^{k-1} - 1; \infty[$.

⋆ Tout d'abord si $n \in N_e$, alors dans le cas (1), $n \geq 1 + 2\,\alpha_{e-1} = \alpha_e = 2^{e-1}$ sinon, dans le cas (2), $n > 1 + \alpha_e$. On aboutit donc dans les deux cas à $n \in [\![\alpha_e; \infty[$.

⋆ Réciproquement, soit n un entier $\geq \alpha_e = 2^{e-1}$. On cherche à décomposer n par exemple en $n = 1 + n_1 + n_2$ avec n_1 et n_2 dans N_{e-1} pour se ramener au cas (1) si c'est possible.

Prenons $n_1 = 2^{e-2} - 1$ et $n_2 = n - n_1 - 1$. Ainsi déjà $n_1 \in N_{e-1}$.

D'autre part, $n_2 \geq (2^{e-1} - 1) - (2^{e-2} - 1) - 1 = 2^{e-2} - 1$, donc $n_2 \in N_{e-1}$.

Par définition de N_{e-1}, il existe deux arbres `A1` et `A2` d'épaisseur $e-1$ de taille respective n_1 et n_2.

Les deux arbres dont la racine a pour fils `A1` et `A2` ont pour épaisseur e et pour taille $1 + n_1 + n_2 = n$, ce qui prouve que $n \in N_e$.

Remarque : l'arbre de taille minimum d'épaisseur $e \geq 2$ est l'arbre parfait de hauteur $h = e - 1$ dont le nombre de noeuds internes est $\alpha_e = 2^{e-1} - 1$.

Question 8

Une structure de données Caml permettant de décrire les arbres "squelettiques" étudiés dans cette partie **I** est par exemple :

```
type arbre_1 = | F                          (* Feuille *)
               | N of arbre_1 * arbre_1 ;; (* Noeud *)
```

Pour implémenter l'exemple donné à la figure **5**, on déclare alors :

```
let exfig5 = N(N(N(F,F),N(N(N(F,N(F,F)),F),F)),N(F,N(F,F))) ;;
```

On constate que l'encombrement mémoire est proportionnel au nombre de noeuds.

Cela ne dépend pas du langage utilisé, car on peut obtenir une bijection entre la représentation Caml précédente et un codage sur l'alphabet formé des deux caractères `(` et `)`.

⋆ L'arbre réduit à un noeud est codé `()`.
⋆ Si `u` est le code de `Ag` et `v` celui de `Ad`, alors le code de l'arbre dont la racine a pour fils gauche `Ag` et pour fils droit `Ag` est codé par le mot `(uv)`.
Ainsi le code de l'arbre de la figure **5** est obtenu à partir de la représentation Caml en supprimant les lettres `N`, `F` et la virgule, ce qui donne : `(((()((((()))))(()))))`.

Inversement, si tout mot "bien parenthésé" représente effectivement un arbre, (ce qu'on peut montrer par induction structuelle) son écriture Caml n'est pas immédiate car il faut pouvoir séparer l'arbre gauche et l'arbre droit.

Question 9

```
let val i j = if i=j then i+1 else max i j;;
let rec epaisseur a = match a with
     | F      -> 1
     | N(g,d) -> val (epaisseur g) (epaisseur d) ;;
```

La fonction `val` coûte soit une comparaison et une addition entière, soit deux comparaisons.
Le calcul de l'épaisseur d'un arbre de taille n s'effectue avec n appels à la fonction `val`, d'où un coût linéaire.

Question 10

On part d'un réel aléatoire $x \in [0,1[$.

Considérons la fonction $r1$ définie sur $[0,1[$ par $r1(x) = \begin{cases} 2\,x & \text{si } x < 0.5 \\ 2\,x-1 & \text{sinon} \end{cases}$

On vérifie facilement que $r1$ est aussi à valeurs dans $[0,1[$.
Cette fonction "d'inspiration dichotomique" permet de construire une suite (x_n) définie par $x_0 = x$ et $x_{n+1} = r1(x_n)$ à laquelle est associée une suite booléenne (b_n) définie par :

$b_n =$ `true` si $x_n < 0.5$ et $b_n =$`false` si $x_n \geq 0.5$.

Ainsi pour mettre un noeud `N(F,F)` à la place d'une feuille `F` dans un arbre de taille $n-1$ pour obtenir un arbre de taille n, on procède comme suit : partant de la racine, si $b_0 =$`true`, alors on continue avec le fils gauche, sinon avec le fils droit, puis on recommence en examinant le booléen b_1 et cela tant que l'on n'a pas atteint une feuille. La fonction Caml `r1` ci-dessous appliquée à x_n renvoie le couple (b_n, x_{n+1}), ce qui évite de faire deux fois le test "$x_n < 0.5$?"

```
let r1 x = if x < 0.5 then (false, 2. *. x) else (true,  2. *. x -. 1.0);;

let construit_arbre_1 n = let s = ref(N(F,F)) in
  let rec insere a x  = match a with
        | F -> N(F,F)
        | N(g,d) -> let (test, y) = r1 x in
                          if test then  N(insere g y, d)  else  N(g, insere d y)
  in for i=2 to n do let x= random__float 1.0 in s := insere !s x  done;
     !s;;
```

Complexité :

La complexité C_k pour passer d'un arbre de taille $k-1$ à un arbre de taille k dépend de la forme de l'arbre et de sa hauteur h_{k-1}.

Le pire des cas est un de ceux évoqués à la question **6** d'un arbre dégénéré pour lequel $C_k = k-1$.

Ainsi dans le pire des cas conduisant à un arbre dégénéré de taille n, on a :

$$C_n = 1 + 2 + \cdots + (n-1) = \frac{n\,(n-1)}{2}.$$

Un cas intéressant (mais est-ce le meilleur ?) est celui qui produit à chaque étape un arbre équilibré (dans lequel toutes les feuilles sont à la hauteur $h-1$ ou h) : on a alors $C_k \equiv_{k\to\infty} \log_2 k$, d'où $C_n \sim \sum_{k=2}^{n} \log_2 k \equiv_{n\to\infty} n \log_2 n$ *(en comparant avec une intégrale).*

Question 11

L'algorithme proposé peut-être traduit en Caml de la façon suivante :

```
let r2 k = 2 + random__int (k-1);;  (* r\'esultat dans [[2;k]] *)

let rec cree_arbre_1 e =
      if e=2 then N(F,F)
      else let i = r2 e in
           if i < e then let a1 = cree_arbre_1 i
                         and a2 = cree_arbre_1 e in N(a1,a2)
                    else let  b = cree_arbre_1 (e-1) in N(b,b)
      ;;
```

Théoriquement, cet algorithme peut boucler indéfiniment du fait de l'appel récursif avec le même paramètre e que celui de la fonction initiale.

La probabilité p_n d'obtenir la valeur k après n appels `r2(k)` est égale à $\left(\frac{1}{k-1}\right)^n$: elle tend donc vers 0 avec n (mais sans être égale à 0).

En pratique, l'algorithme s'arrête "presque toujours", mais peut conduire à des arbres de taille quelconque, parfois trés grande, conformément au résultat de la question **7**.

Question 12

Dans ce qui précède, on a tenu compte uniquement de l'épaisseur e de l'arbre à construire sans se préoccuper de la matrice de ramification en faisant intervenir une unique fonction `r2`.

Si l'on admet qu'on puisse construire des fonctions aléatoires $\texttt{r2}_k$ dépendant de chaque ligne de la matrice, alors la méthode précédente qui est probabiliste, donc non déterministe conduira à l'obtention d'un arbre d'épaisseur égale à e, mais dont la matrice de ramification sera seulement "voisine" de la matrice donnée à l'avance.

Les formes arborescentes obtenues seront trés variées, ce qui est normal puisque des arbres de même épaisseur peuvent avoir la même matrice de ramification, ce qui explique qu'on ait recours à une méthode aléatoire pour constuire l'un d'eux, ou du moins un qui s'en rapproche.

Cependant les différentes solutions obtenues en appliquant plusieurs fois de suite cette méthode pour une matrice de ramification donnée seront toutes plutôt effilées si la matrice est bien remplie inférieurement, ou toutes plutôt touffues si la matrice est creuse.

Complément :

Le programme suivant (non demandé) permet de calculer la matrice de ramification de n'importe quel arbre binaire strict et de vérifier les réponses des questions **1, 3** et **4**.

Les fractions sont représentées par des couples d'entiers.

```
let matrice a =
  let e = epaisseur a in    (* premier parcours n\'ecessaire pour *)
                            (* initialiser les matrices        *)
  let v = make_vect e 0 and m = make_matrix e e 0 in
     let rec parcours aa = match aa with
        | F        -> 1;
        | N(g,d)   -> let i = parcours g and j = parcours d in
                if i <= j then  m.(i-1).(j-1) <- m.(i-1).(j-1) + 1
                          else  m.(j-1).(i-1) <- m.(j-1).(i-1) + 1;
                let k = val i j in v.(k-1) <- v.(k-1) + 1;
                k;
     in parcours a;

     let r = make_matrix e e (0,0) in
         r.(0).(0) <- (1,1);      (* fractions repr\'esent\'ees par des couples *)
         for k=1 to e-1 do
            r.(k).(k) <-  (m.(k-1).(k-1) , v.(k));
            for i=0 to k-1 do r.(k).(i) <- if m.(i).(k)= 0 then (0,0)
                                           else (m.(i).(k),v.(k)) done
         done;
         r;;

(* matrice exfig5;;            -->     - : (int * int) vect vect =
 [|[|1, 1; 0, 0; 0, 0|]; [|4, 7; 3, 7; 0, 0|]; [|0, 0; 1, 2; 1, 2|]|] *)

let rec parfait_1 h = if h=1 then N(F,F) else let s = parfait_1 (h-1) in N(s,s);;

let rec peigne_1 h = if h=1 then N(F,F) else  N(F, peigne_1 (h-1));;

(* matrice (parfait_1 3) ;;    -->    - : (int * int) vect vect =
 [|[|1, 1; 0, 0; 0, 0; 0, 0|]; [|0, 0; 4, 4; 0, 0; 0, 0|];
   [|0, 0; 0, 0; 2, 2; 0, 0|]; [|0, 0; 0, 0; 0, 0; 1, 1|]|]          *)

(* matrice (peigne_1 5) ;;     -->
- : (int * int) vect vect = [|[|1, 1; 0, 0|]; [|4, 5; 1, 5|]|]       *)
```

Deuxième partie : une application de la notion d'épaisseur

Les registres servent d'une part à stocker les résultats de l'évaluation de sous-expressions.
D'autre part, un terme figurant dans une expression n'est mis dans un registre qu'au moment précis où il intervient dans une opération.

Question 13

$R_0 \leftarrow a$; $R_1 \leftarrow b$; $R_2 \leftarrow c$; $R_1 \leftarrow R_1 - R_2$; $R_2 \leftarrow d$; $R_3 \leftarrow e$;
$R_2 \leftarrow R_2 - R_3$; $R_1 \leftarrow R_1 * R_2$; $R_0 \leftarrow R_0 + R_1$.

On a donc utilisé 4 registres.

Question 14

$R_0 \leftarrow e$; $R_1 \leftarrow d$; $R_0 \leftarrow R_1 - R_0$; $R_1 \leftarrow c$; $R_2 \leftarrow b$; $R_1 \leftarrow R_2 - R_1$;
$R_0 \leftarrow R_0 * R_1$; $R_1 \leftarrow a$; $R_0 \leftarrow R_0 + R_1$.

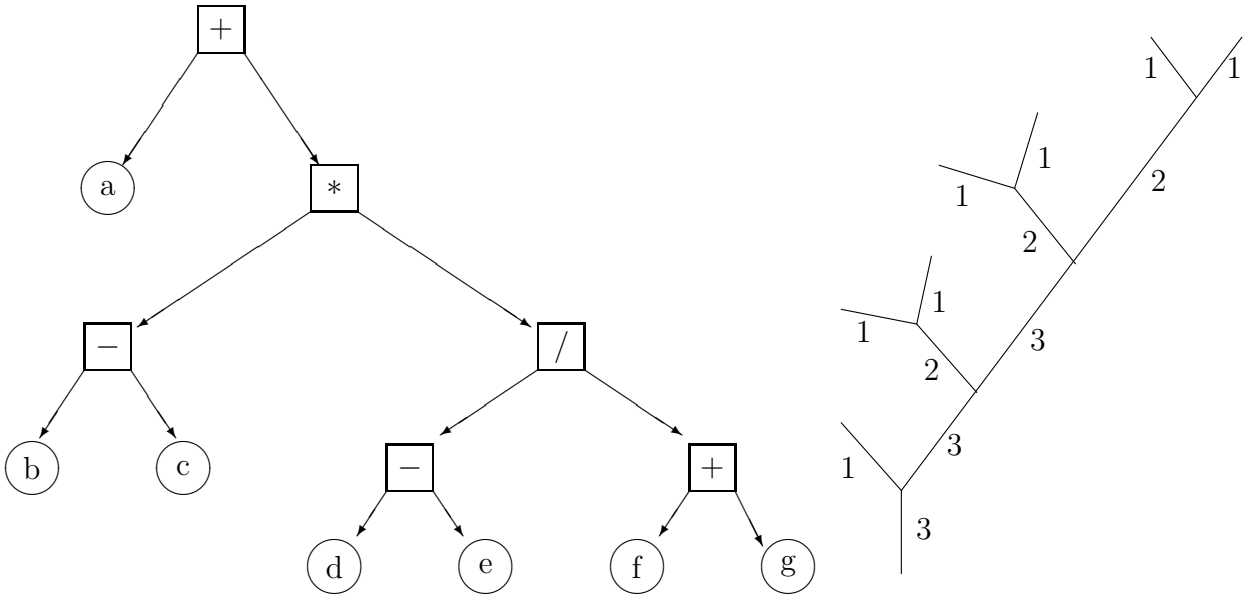
On a donc utilisé 3 registres.

Question 15

On évalue d'abord le 2^e terme en appliquant l'associativité à gauche ($*$ et / étant de même priorité).
$R_0 \leftarrow b$; $R_1 \leftarrow c$; $R_0 \leftarrow R_0 - R_1$; $R_1 \leftarrow d$; $R_2 \leftarrow e$; $R_1 \leftarrow R_1 - R_2$; $R_0 \leftarrow R_1 * R_2$;
$R_1 \leftarrow f$; $R_2 \leftarrow g$; $R_1 \leftarrow R_1 + R_2$; $R_0 \leftarrow R_0/R_1$; $R_1 \leftarrow a$; $R_0 \leftarrow R_0 + R_1$.

On a donc utilisé 3 registres.

Question 16



Question 17

On peut définir un type Caml d'expression infixée de la façon suivante :

```
type Expression = | Var of char | Oper of Expression * char * Expression ;;
```

A toute expression $Expr$, on associe un arbre de la façon suivante :

- si $Expr$ est une variable, alors c'est l'arbre réduit à `F`.
- si $Expr$ est de la forme $Expr1 * \omega * Expr2$, alors c'est l'arbre $A = N(A1, A2)$ où $A1$ (resp . $A2$) est l'arbre associé à $Expr1$ (resp. $Expr2$).

On peut alors établir par **induction structurelle** que la stratégie optimale d'évaluation de $Expr$ utilise un **nombre de registres égal à l'épaisseur de l'arbre associé.**

- C'est vrai si $Expr$ est une variable en convenat qu'une feuille est d'épaisseur 1 comme sur les schémas.
- Supposons que $Expr$ est de la forme $Expr1 * \omega * Expr2$.
  Notons $A1$ et $A2$ les arbres associés à $Expr1$ et $Expr2$ respectivement, n_1 et n_2 leus épaisseurs.
  L'hypothèse de récursion est que que n_1 (resp. n_2) est égal au nombre total de registres utilisés (avec la stratégie optimale) pour l'évaluation de $Expr1$ (resp. $Expr2$).

  $\star$ si $n_1 < n_2$ (nécessairement $n_2 \geq 2$), alors on commence par évaluer $Expr2$, ce qui utilise n_2 registres et on stocke le résultat dans R_1. Puis on évalue $Expr1$ à l'aide des registres $R_2, \ldots, R_{n_1+1}$, le résultat étant placé dans R_2.
  Il a donc fallu $\max\{n_2, n_1 + 1, 2\} = n_2$ registes pour évaluer $Expr$.
  Or l'épaisseur de l'arbre A associé à $Expr$ est aussi dans ce cas $n = n_2$.

  $\star$ si $n_2 < n_1$, résultat analogue en commençant par évaluer $Expr1$.

  $\star$ si $n_1 = n_2$, alors on commence par exemple, comme Caml, par évaluer $Expr2$, ce qui utilise n_2 registres et on stocke le résultat dans R_1. Puis on évalue $Expr1$ à l'aide des registres $R_2, \ldots, R_{n_1+1}$, le résultat étant placé dans R_2.
  Il a donc fallu $n_1 + 1$ registes pour évaluer $Expr$.
  Or l'épaisseur de l'arbre A associé à $Expr$ est aussi dans ce cas $n = n_1 + 1$.

Complément :

Caml évaluant les arguments de ces fonctions de droite à gauche, il peut être utile de se ramener à un arbre dans lequel chaque fils droit ait une épaiseur supérieure ou égale à celle de son fils gauche.
C'est ce que réalise la fonction ci-dessous :

```
let optimise a =
       let rec parcours aa = match aa with
        | F        -> (1,F);
        | N(g,d)   -> let (i,ag) = parcours g and (j,ad) = parcours d in
                          let k = val i j in
                                 if i <= j then (k,N(ag,ad)) else (k,N(ad,ag))
     in let (e,a') = parcours a in a'
   ;;

(* optimise exfig5;;         -->          - : arbre_1 =
 N (N (F, N (F, F)), N (N (F, F), N (F, N (F, N (F, N (F, F))))))      *)
```

Troisième partie : codages d'arbres binaires

Je me demande encore à quoi peut bien servir la définition de mot bien emboité donné par l'énoncé, car il ne caractérise pas les codes d'arbres binaires obtenus dans cette partie.
Certes, tout code est un mot bien emboité, mais la réciproque est fausse car, par exemple, `[a][a]` *n'est pas le code d'un arbre.*
Il semble qu'il y ait une erreur d'énoncé à la question 20 *où il fait prendre les* x_i *dans* Σ^+*. Dans la suite, on ne tient pas compte de quel coté de l'axe les branches sont dessinées.*

On peut définir un autre type d'arbre Caml pour écrire les algorithmes étudiés dans cette troisième partie où cette fois un noeud peut être de degré 0 ("feuille") ou 1 si l'un des fils est "vide".
Le codage d'un tel arbre ne pose pas de problème, en convenant que le fils gauche éventuel est une branche et que le fils droit est situé sur l'axe.

```
type arbre_3 =  | V       (* Vide *)
                | F       (* Feuille *)
                | N of arbre_3 * arbre_3;;

let rec code_arbre a = match a with
     |     V    -> ""
     |     F    -> "a"
     | N(V,f2)  ->  "a"  ^ (code_arbre f2)
     | N(f1,f2) ->  "a[" ^ (code_arbre f1) ^ "]" ^ (code_arbre f2)
  ;;
```

Question 18

C'est bien sûr `aa[aa]aa[a[a]a]a[aaa]aa`.

Question 19

Plusieurs réponses sont possibles pour caractériser les branches.

1) Sachant que `u` représente le codage de l'arbre (donc est correct), quand on extrait un sous-mot de la forme `[w]`, pour que `w` représente une branche, il faut et il suffit que `w` contienne autant de crochets droits que de crochets gauches.

2) Si on veut être plus précis (ce qui servira pour la question **20**), on peut donner la caractérisation suivante :

- ⋆ tout crochet (gauche et droit) figurant dans `w` est précédé et suivi d'un `a`.
- ⋆ tout préfixe de `w` contient un nombre de crochets gauches supérieur ou égal à celui des crochets droits.
- ⋆ `w` contient autant de crochets gauches et droits.

L'intérêt de cette caractérisation est de pouvoir servir à l'analyse syntaxique des mots de Σ_E^*.

Question 20

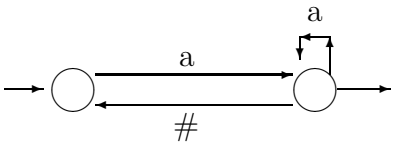
Considérons l'énoncé $\mathcal{E}_k$ suivant :
"Toute branche contenant k noeuds internes admet un codage $[w]$ qui se décompose sous la forme :
$[w] = [x_1[\alpha_1]x_2 \dots [\alpha_n]x_{n+1}]$ où les $x_i \in \boxed{\Sigma^+}$ et les $[\alpha_i]$ sont eux-mêmes des codages de branches."
Remarque :
l'unicité d'une telle décomposition provient de la deuxième caractérisation donnée à la question **19**.
Si $k = 1$, alors il s'agit d'un arbre parfait de hauteur 1 qui est codé : $a[a]a$, donc $\mathcal{E}_1$ est vrai.
Supposons les énoncés $\mathcal{E}_1, \dots, \mathcal{E}_{k-1}$ vrais.
Considérons une branche b ayant k noeuds internes. Deux cas se présentent :

⋆ si la racine de b est un noeud de degré 1, alors le fils unique b_1 de b est une branche contenant $k-1$ noeuds internes, qui se code en $[w_1] = [x_1[\alpha_1]\dots[\alpha_n]x_{n+1}]$, et b admet alors pour codage $[w] = [aw_1] = [ax_1[\alpha_1]\dots[\alpha_n]x_{n+1}]$.

⋆ si la racine de b est un noeud de degré 2, alors chaque fils b_1 et b_2 de b est une branche contenant au plus $k-2$ noeuds internes, l'axe de b contenant par exemple celui de b_2.
D'après l'hypothèse de récurrence, b_1 se code avec $[w_1] = [x_1[\alpha_1]\dots[\alpha_p]x_{p+1}]$ et b_2 avec $[w_2] = [y_1[\beta_1]\dots[\beta_q]y_{q+1}]$.
Alors b est codé par $[w] = [a[w_1]w_2] = [a[w_1]y_1[\beta_1]\dots[\beta_q]y_{q+1}]$.

Question 21

Il s'agit des mots commençant par un `a`, se terminant par un `a` et dans lesquels deux caractères # sont toujours séparés par au moins un `a`.
Une expression rationnelle de ce langage est $a\,(a + \#a)^*$ et un AFD le reconnaissant est :



Question 22

$w_1 = a[a]a$ est le code de l'arbre parfait de hauteur 1.
Si w_{n-1} est le code de l'arbre parfait de hauteur $n-1$, alors le code de celui de hauteur n est $\boxed{w_n = a[w_{n-1}]w_{n-1}.}$ On vérifie facilement par récurrence que $|w_n| = 2^{n+2} - 3$.
On peut convenir que l'arbre parfait de hauteur 0 (c'est une feuille) existe est est codé par a.
Il est facile de programmer le calcul de w_n :

```
let rec cree_code_parfait n =
  if n=0 then "a"
         else let s = cree_code_parfait (n-1) in "a[" ^ s ^ "]" ^ s
  ;;
```

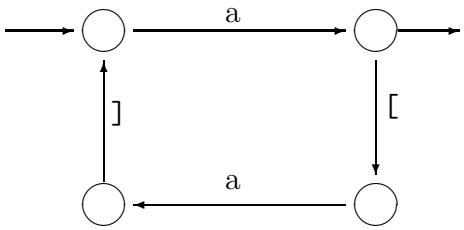
Pour tester si un mot w est le codage d'un arbre parfait, on peut procéder récursivement de la façon suivante :
- si $|w| = 1$, alors w est parfait si et seulement si $w =$ "a".
- sinon soit m la longueur de w. On forme $p = \lfloor (m-3)/2 \rfloor$ et on extrait le sous-mot u commençant à l'indice 2 et se terminant à l'indice $(p+1)$ et le sous-mot v commençant à l'indice $(p+3)$ et se terminant à l'indice $(m-1)$.
Alors w est parfait si et seulement si
  . w commence par "a["
  . le caractère d'indice $p+2$ est un crochet droit
  . les deux sous-mots u et v sont égaux
  . v est un mot parfait *(un seul appel récursif).*

En voici une implémentation en Caml (sans utiliser la fonction `sub_string`) :

```
let test_code_parfait w =
  let rec  egal i  k  p = (p = 0) || (w.[i]=w.[k] && egal (i+1) (k+1) (p-1) )
  in let rec aux i j =
       if i=j then w.[i]='a'else  let p = (j-i-2) / 2 in
          w.[i]='a'     &&  w.[i+1]= '['       && w.[i+p+2]= ']' &&
          p = j-i-p-2   &&  egal (i+2) (i+p+3) p && aux (i+p+3) j
  in aux 0 (string_length w -1)
  ;;
```

Question 23

Les codes caractérisant les arbres peignes sont les mots de la forme $a[a]a[a]a\ldots[a]a$, le nombre de couples de crochets étant égal à la hauteur de l'arbre.
Une expression rationnelle de ce langage est $a\,([a]a)^*$ et un AFD le reconnaissant est :.



Question 24

Les codages de formes arborescentes contenant un nombre variable de crochets emboités ne sont pas reconnaissables par automate fini.

Un algorithme itératif permettant de dire si un mot w est le code d'une forme arborescente consiste à lire w de gauche à droite en vérifiant que :
- chaque crochet est précédé et suivi d'au moins un a.
- tout préfixe de w contient au moins autant de crochets gauches que de crochets droits.
- w contient autant de crochets gauches que de crochets droits.

Dans les deux fonctions Caml qui suivent, l'une itérative, l'autre récursive, i représente l'indice courant de parcours dans la chaîne et c l'excédent de crochets gauches.
Remarque : Si l'on attribue les poids $0,\ 1,\ -1$ respectivement à $a,\ [,\ ]$, alors c est le poids du préfixe de w de longueur $i-1$, ce qui fournit un invariant de boucle.

```
let test_code_iter w =
    let n = string_length w and i = ref 0 and  c = ref 0 in
    while !i < n-1  do match (w.[!i], w.[!i+1]) with

            | ‘a‘,‘a‘  ->  i:= !i+1
            | ‘a‘,‘[‘  ->  c := !c+1; i:= !i + 2;
            | ‘a‘,‘]‘  ->  if !c = 0 then failwith "Il manque un ["
                           else begin c := !c-1; i := !i+2 end
            | ‘a‘,_    ->  failwith "Caract\‘ere non autoris\’e "
            |    _     ->  failwith "Il manque un a"
    done;
    if w.[n-1] <> ‘a‘ then failwith "Il manque le a final"
                      else !c = 0 ;;

let test_code_rec1 w =
  let n = string_length w in
  let rec aux i c =
           (i= n  &&  c=0)
        || (i=n-1 &&  c=0 &&  w.[i] = ‘a‘ )
        || match (w.[i], w.[i+1]) with
               | ‘a‘,‘a‘  ->  aux (i+1) c
               | ‘a‘,‘[‘  ->  aux (i+2) (c+1)
               | ‘a‘,‘]‘  ->  if c=0 then failwith "Il manque un ["
                              else aux (i+2) (c-1)
               | ‘a‘,_    ->  failwith "Caract\‘ere non autoris\’e "
               |    _     ->  failwith "Il manque un a"
  in aux 0 0  ;;
```

Une autre façon de programmer récursivement qui évite le compteur c consiste à lire la chaîne de droite à gauche.

```
let test_code_rec2 w =
  let i = ref (string_length w - 1) in
  let rec aux () =
     if !i < 0 then failwith " Incorrect"
     else if !i = 0 then w.[0]=‘a‘
          else match (w.[!i-1], w.[!i]) with
                | ‘a‘,‘a‘  ->  i := !i-1; aux()
                | ‘]‘,‘a‘  ->  i := !i-2; aux();aux()
                | ‘[‘,‘a‘  ->  i := !i-2; aux()
                | ‘_‘,‘a‘  ->  failwith "Caract\‘ere non autoris\’e "
                |    _     ->  failwith "Il manque un a"
  in aux ()
  ;;
```

En adaptant l'une des fonctions précédentes, on obtient facilement la forme arborescente associée à un mot w ("correct") donné.

```
let arbre_du_code w =
  let i = ref (string_length w - 1) in
  let rec aux a2 =
      if !i < 0 then  failwith "Incorrect"
      else if !i = 0 then if w.[0]=‘a‘ then a2
                          else failwith "Erreur"
           else match (w.[!i-1], w.[!i]) with
                | ‘a‘,‘a‘  ->  i := !i-1;  aux  (N(V,a2))
                | ‘]‘,‘a‘  ->  i := !i-2; let b=aux F in aux (N(b,a2))

                | ‘[‘,‘a‘  ->  i := !i-2; a2
                | ‘_‘,‘a‘  ->  failwith "Caract\‘ere non autoris\’e "
                |    _     ->  failwith "Il manque un a"
  in aux F
  ;;
```

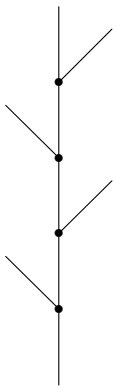
Question 25

Plusieurs conventions de dessins d'arbres peuvent être choisies, l'énoncé n'en ayant fixé aucune.
Si l'on observe l'exemple proposé à la figure 10, on remarque que les branches sont disposées alternativement à gauche et à droite de l'axe.
En outre, dans une branche placée à gauche par exemple, la première sous-branche est elle aussi placée à gauche.
Ce sera le rôle des booléens `sens` et `first` de permettre d'appliquer cette convention.
Toutes les arêtes sont représentées par des vecteurs de longueurs "presque égales" dans 8 directions (tous les 45^o) stockées dans un tableau `dir` défini en variable globale.

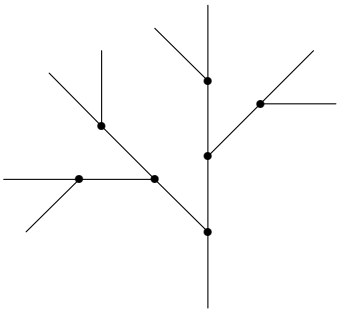
La fonction Caml `mod` ne donnant pas le résultat escompté pour les entiers négatifs, la fonction `modulo` ci-dessous renvoie un reste positif ou nul.

La fonction `dessine` permet de dessiner un arbre, la racine étant placée en (x_0, y_0) avec le facteur d'échelle entier k.

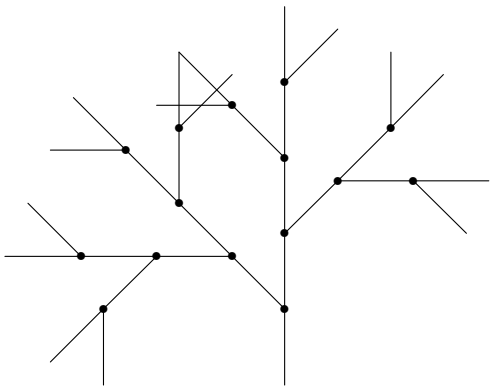
```
let dir = [|(0,10);(7;,7);(10,0);(7,-7);
           (0,-10);(-7,-7);(-10,0);(-7,7)|];;

let rec modulo x n = if x >=0 then x mod n else modulo (x+n) n;;

#open "graphics";;
open_graph "";;

let dessine a k x0 y0 =                (*  k = facteur entier d'\'echelle    *)
  clear_graph();                       (* (x0,y0) = position de la racine  *)
  tortue a x0 y0 0 true true where     (*  celle-ci \'etant plac\'ee en bas    *)

   rec tortue aa x y i sens first = match aa with
      |    V     -> ()
      |    F     -> moveto x y ;
                    let (dx,dy)=dir.(modulo i 8) in lineto (x+k*dx) (y+k*dy)
      | N(f1,f2) -> moveto x y ;
                     let (dx,dy)=dir.(modulo i 8) in
                     let x1=x+k*dx and y1=y+k*dy in
                         lineto x1 y1;
                         fill_circle x1 y1 2;
                         tortue f2 x1 y1 i (not sens) true;
                     if first then
                        if sens then tortue f1 x1 y1 (i-1)  sens  true
                                else tortue f1 x1 y1 (i+1)  sens  true

                             else
                        if sens then tortue f1 x1 y1 (i+1) (not sens) false
                                else tortue f1 x1 y1 (i-1) (not sens) false

   ;;
```



Peigne : h=4



Parfait : h=3



Parfait : h=4

$*+*+*+*+*+*+*+*$