EPREUVE D'INFORMATIQUE (Sujet commun ENS : ULM et LYON) 1997

Première Partie

Question 1

Un petit comptage fournit la matrice de ramification ci-contre : $\left(1\ 0\ 0\ \frac{4}{7}\ \frac{3}{7}\ 0\ 0\ \frac{1}{2}\ \frac{1}{2}\right)$

Question 2

Soit k^32. D'après la remarque du bas de la page 4/10 de l'énoncé, on a $N_t(k) = \sum_{i=1}^{k-1} N_t(i,k) + N_t(k-i, k-1)$.

Donc $\sum_{i=1}^{ep(t)} R_{k,i} = \sum_{i=1}^{ek} R_{k,i} = 1$. Comme de plus $\sum_{i=1}^{ep(t)} R_{1,i} = R_{1,1} = 1$, on a finalement :

La somme des éléments d'une ligne de la matrice de ramification vaut toujours 1

Question 3

Un arbre binaire strict est parfait si et seulement si il est réduit à une feuille, ou si ses deux sous-arbres sont parfaits et de même hauteur. Par conséquent, il suffit de prouver par récurrence la propriété suivante sur l'entier h :

(un arbre binaire strict de hauteur h est parfait) $\Leftrightarrow$ (tous les chemins de la racine à une feuille ont pour longueur h)

- La propriété est vraie pour h = 0, car un arbre de hauteur nulle est réduit à une feuille, donc est parfait et ne contient aucune arête.
- Supposons la propriété vraie au rang $h - 1$, et considérons un arbre parfait de hauteur h. Ses deux branches sont des arbres parfaits de hauteur $h - 1$, donc tous les chemins menant des racines de ces branches à leurs feuilles ont pour longueur $h - 1$. Mais tout chemin menant de la racine de l'arbre à une de ses feuilles consiste à traverser l'une des deux arêtes menant à la racine d'un sous-arbre, puis à prendre un chemin (de longueur $h - 1$) menant de là à une feuille de ce sous-arbre. Il est donc de longueur $h - 1 + 1 = h$.
  Réciproquement, si tout chemin menant de la racine de l'arbre à une de ses feuilles est de longueur h^31, alors cet arbre possède deux sous-arbres de hauteur inférieure ou égale à $h - 1$, et tout chemin menant de la racine de l'un de ces sous-arbres vers une feuille est de longueur $h - 1$, ce qui prouve que ces deux sous-arbres sont effectivement de hauteur $h - 1$, car ils contiennent au moins une feuille. Par hypothèse de récurrence, les deux sous-arbres sont donc parfaits. Comme ils ont même hauteur $h - 1$, on en déduit que l'arbre entier est parfait.

un arbre binaire strict est parfait si et seulement si tout chemin de la racine à une feuille a pour longueur sa hauteur

Les sous-arbres d'un arbre parfait de hauteur h étant parfaits, on montre facilement par récurrence que tous les noeuds internes d'un même niveau ont même biépaisseur, ce qui conduit à une épaisseur finale de l'arbre égale à h + 1, d'où la taille de la matrice de ramification. D'autre part, il n'existe aucun noeud interne dont la biépaisseur est de type (i,j), avec i^1j. Donc la matrice est diagonale, ne comportant que des 1 sur la diagonale.

La matrice de ramification d'un arbre parfait de hauteur h est la matrice identité d'ordre h + 1

Question 4

La matrice de ramification d'un arbre-peigne de hauteur 1 est $(1\ 0\ 0\ 1)$.

Montrons par récurrence que si $h^3 2$, la matrice d'un tel arbre vaut $\left(1\ 0\ \frac{h-1}{h}\ \frac{1}{h}\right)$. C'est clairement vrai si $h = 2$.

Supposons que c'est vrai pour un arbre-peigne de hauteur $h^3 2$. Alors le sous-arbre droit d'un arbre-peigne de hauteur h + 1 est un arbre-peigne de hauteur h, tandis que son sous-arbre gauche est un arbre-feuille. Par hypothèse de récurrence, la matrice de ramification du sous-arbre droit est de taille 2, donc le sous-arbre droit est d'épaisseur 2. Comme le sous-arbre gauche est d'épaisseur 1, l'arbre entier est d'épaisseur 2. De plus, les noeuds internes du sous-arbre droit se répartissent en h − 1 noeuds de biépaisseur (1,2) et 1 noeud de biépaisseur (1,1), ce qui fournit pour l'arbre entier h noeuds de biépaisseur (1,2) et 1 de biépaisseur (1,1), soit la matrice de ramification $\left(1\ 0\ \frac{h}{h+1}\ \frac{1}{h+1}\right)$.
En remarquant que la forme générale est aussi valable pour h = 1, on peut conclure :

$$\underline{Pour\ h^3 1,\ la\ matrice\ de\ ramification\ d©un\ arbre\ -\ peigne\ de\ hauteur\ h\ vaut\ \left(1\ 0\ \frac{h-1}{h}\ \frac{1}{h}\right)}$$

Question 5

La croissance de la taille de la matrice de ramification d'un arbre exige une épaisseur identique des deux sous-arbres. Cette matrice sera donc de taille d'autant plus grande qu'à chaque profondeur les sous-arbres seront mieux « équilibrés », le nec plus ultra en la matière étant l'arbre parfait.
En contrepartie, le nombre de noeuds ayant une biépaisseur dont le premier terme est très inférieur au second sera plus limité, et les valeurs les plus grandes prises par les termes de la matrice seront resserrées « autour de la diagonale » ; on peut aussi s'attendre à trouver une matrice de ramification d'autant plus creuse (avec des zéros) qu'elle est de grande taille. En résumé :

<u>La matrice de ramification d'un arbre est d'autant plus grande et proche d'une matrice diagonale que l'arbre est plus symétrique. A l'inverse, un arbre asymétrique aura une matrice de petite taille par rapport à sa hauteur et bien pleine</u>

Question 6

D'après l'exemple caricatural de la question 4, on a $n^3 1, \lambda\ left\ (n\ right\ ) = 2\}\{$.
D'autre part, pour qu'un arbre de taille $n^3 2$ soit d'épaisseur 2, il faut et il suffit que l'un de ses deux sous-arbres soit de taille 1 et l'autre de taille 2, donc le nombre u_n d'arbres de taille 2 vérifie les relations $\{$. D'où :

$$Le\ nombre\ d©arbres\ de\ taille\ n^3 1\ et\ d©paisseur\ \lambda(n)\ est\ 2^{n-1}$$

Question 7

Soit $h^3 2$. Pour qu'un arbre soit d'épaisseur h, il est nécessaire qu'il contienne quelque part deux arêtes d'épaisseur h − 1 issues d'un même noeud interne.
Appelons v_h le nombre <u>minimal</u> de noeuds d'un arbre d'épaisseur h. On a donc $v_1 = 0$et $h^3 2, v\ rSub\{\ size\ 8\{h\ +\ 1\}\}\ ^3 2 v\ rSub\{\ size\ 8\{h\}\}\ +\ 1\}\{$(+1 pour tenir compte du noeud de confluence). On montre alors aisément par récurrence que $h^3 2, v\ rSub\{\ size\ 8\{h\}\}\ ^3 2\ rSup\{\ size\ 8\{h\ -\ 1\}\}\ -\ 1\}\{$.
Réciproquement, montrons par récurrence que pour tout entier $n^3 2^{h-1} - 1$, il existe un arbre de taille n et d'épaisseur h ($h^3 2$).

- pour $n = 2^{h-1} - 1 = \sum_{k=0}^{h-2} 2^k$, il suffit de considérer un arbre parfait de hauteur h − 1.
- supposons construit un arbre de taille $n^3 2^{h-1} - 1$ et d'épaisseur h. Alors l'arbre dont le sous-arbre gauche est une feuille et le sous-arbre droit est l'arbre ainsi construit possède n + 1 noeuds internes et conserve une épaisseur égale à h, car la biépaisseur de son tronc est (1,h), avec $h^3 2$.

Un arbre d©paisseur h peut avoir toute taille suprieure ou gale ˆ $2^{h-1} - 1$

Question 8

Une forme arborescente étant caractérisée par un arbre binaire strict, on peut bien évidemment utiliser le Type de Données Abstrait arbre binaire pour représenter une telle forme. Rappelons que ce TDA est défini récursivement par:

arbre binaire = feuille + arbre binaire x noeud interne x arbre binaire

La spécifité de la représentation d'une forme arborescente réside dans la définition du noeud interne : il convient d'y stocker l'épaisseur (ou la biépaisseur) de l'arête menant à ce noeud, ainsi que d'autres informations telles que la longueur de cette arête et l'angle qu'elle fait avec une direction fixe. La meilleure structure de données pour définir un tel noeud est bien évidemment un enregistrement comportant autant de champs que nécessaire (on peut aussi par exemple intégrer un champ couleur).
L'encombrement mémoire pour un noeud est donc proportionnel au nombre des paramètres dont on choisit de tenir compte, et il en résulte que l'encombrement mémoire d'une forme arborescente est proportionnel à sa taille.

Le problème de l'encombrement mémoire est indépendant du langage de programmation utilisé, bien que la définition de la SDD effective puisse subir quelques variations d'un langage à l'autre. Voici un exemple d'implémentation des formes arborescentes en Caml :

```
type forme_arborescente = Feuille | Tronc of
forme_arborescente*noeud*forme_arborescente
and noeud = {mutable épaisseur : int ; mutable longueur : float ; mutable angle :
float};;
```

Question 9

On utilise un algorithme récursif fondé sur la définition de l'épaisseur donnée en page 4/10 :

```
ALGO épaisseur(f : forme_arborescente) → entier
DEBUT
        SI f est une feuille ALORS 1
        SINON
                Ed ← épaisseur(sous_arbre_droit(f))
                Eg ← épaisseur(sous_arbre_gauche(f))
                SI Ed < Eg ALORS Eg
                SINON
                        SI Ed > Eg ALORS Ed SINON 1 + Ed FIN SI
                FIN SI
        FIN SI
FIN
```

```
let rec épaisseur = function
   Feuille -> 1
  |Tronc (g,n,d) ->
    let Ed=épaisseur g and Eg=épaisseur d
    in if Ed<>Eg then max Ed Eg else
1+Eg;;

(* la fonction max simplifie un peu
   l'écriture du programme         *)

épaisseur : forme_arborescente->int =
<fun>
```

L'instruction : SI *Ed* < *Eg* ALORS *Eg* SINON SI *Ed* > *Eg* ALORS *Ed* SINON 1 + *Ed* FIN SI FIN SI est de complexité constante *t*. Donc $\$\beta > 0, 0 \leq t \leq \beta$.
Appelons *T*(*f*) la complexité du calcul de l'épaisseur d'une forme arborescente *f*.
Soit *a* la complexité (constante) du calcul de l'épaisseur d'une feuille et $b = a + \frac{3\beta}{2}$.
La propriété $P(f)$: $taille\ (f) = n \Rightarrow a(n+1) \leq T(f) \leq b(n+1) - \beta$ est donc vraie si f est une feuille, donc de taille nulle.
Soit une forme arborescente f non réduite à une feuille, de taille $n^3 1$, et supposons la propriété vraie pour ses deux sous-branches.

Soit k la taille de la branche gauche g de notre arbre, et par suite $n-1-k$ celle de sa branche droite d.

On a donc { , et comme $T(f) = T(g) + T(d) + t$, on obtient $a(n+1) \leq T(f) \leq b(n+1) - 2\beta + \beta$, d'où $a(n+1) \leq T(f) \leq b(n+1) - \beta$, ce qui prouve la validité de la propriété pour f.
D'après le théorème d'induction sur les arbres binaires stricts :
$f, taille \backslash(f \backslash) = n\, drarrow\, a\, left\,(n+1\, right) <= T\, left\,(f\, right) <= b\, left\,(n+1\, right) - \beta\}\{.$
Donc $\frac{T(f)}{taille(f)}$ est une fonction bornée de f, ce qui se traduit par :

La complexité de l'algorithme du calcul de l'épaisseur d'une forme arborescente est linéaire par rapport à sa taille

Question 10

L'algorithme demandé consiste donc en la construction d'une forme arborescente « ergodique » à l'aide d'un générateur de nombres aléatoires tel que la fonction `random__float` de Caml. L'algorithme de construction peut être construit récursivement, en l'appelant avec une taille $n-1$, puis en développant l'une des feuilles de la forme obtenue pour obtenir un n-ième noeud. Le problème réside dans la façon de choisir aléatoirement l'une des feuilles de la forme arborescente afin de la transformer en noeud interne.
Une solution consiste à sélectionner un chemin de la racine de l'arbre jusqu'à l'une de ses feuilles de la manière suivante : à chaque noeud rencontré, on appelle `r1` et on décide de descendre dans le sous-arbre gauche si le résultat est inférieur à 0,5 et dans le sous-arbre droit sinon. Lorsqu'on arrive à une feuille (cela se produit nécessairement puisque le nombre de noeuds rencontrés au cours de la descente est majoré par n, ce qui assure la terminaison de l'algorithme), on remplace cette feuille par un arbre réduit à un noeud. Le sous-algorithme de descente peut s'écrire de façon récursive.
En voici directement une version en Caml (les champs des noeuds ne sont pas calculés):

```
let rec construit_forme1 n =
   let r1 = random__float 1.
   and germe = Tronc (Feuille,{épaisseur = 1 ; longueur = 0. ; angle = 0.},Feuille)
   in
   let rec développe = function
      Feuille -> germe
     |Tronc (g,noeud,d) ->
       if r1 < 0.5 then Tronc (développe g,noeud,d) else Tronc (g,noeud, développe d)
   in
   match n with
      1 -> germe
     |_ -> développe (construit_forme1 (n - 1));;
```

Le principe de l'algorithme récursif `construit_forme1` est le suivant : si $n = 1$, on retourne un arbre racine. Si $n > 1$, on appelle la fonction sur $n-1$, et on applique au résultat le sous-algorithme récursif `développe`.

1. Distribution des formes construites par `construit_forme1` en fonction de la taille du sous-arbre gauche

   Soit $n \geq 1$, soit k entier relatif, et soit $p_{n,k}$ la probabilité qu'une forme de taille n construite à l'aide de `construit_forme1 n` ait une branche gauche de taille k. On obtient une forme de taille $n+1$ ayant une branche gauche de taille k en appliquant le sous-algorithme `développe`, soit à une forme de taille n ayant une branche gauche de taille $k-1$ lorsque `r1 < 0.5` (ce qui a une probabilité $\frac{1}{2}p_{n,k-1}$ de se produire), soit à une forme de taille n ayant une branche gauche de taille k lorsque `r1 >= 0.5` (ce qui a une probabilité $\frac{1}{2}p_{n,k}$ de se produire). On a donc $p_{n+1,k} = \frac{1}{2}p_{n,k} + \frac{1}{2}p_{n,k-1}$. Il en résulte que la suite $u_{n,k} = 2^{n-1}p_{n,k}$ vérifie la relation de récurrence $n^3 1, k\hat{I}Z, u_{n+1,k} = u_{n,k} + u_{n,k-1}$.
   Mais il est clair que si $k\ddot{I}[0; n-1]$, on a $u_{n,k} = p_{n,k} = 0$. De plus $u_{1,0} = 1$. On en déduit par récurrence,à

l'aide de la Formule de Pascal, que
$k\hat{I}\ left\ [0; n\ -\ 1\ right\ ], u\ rSub\ \{\ size\ 8\{n, k\}\ \}\ =\ C\ rSub\ \{\ size\ 8\{n\ -\ 1\}\ \}\ rSup\ \{\ size\ 8\{k\}\ \}\ \}\ \{$, puis que
$k\hat{I}\ left\ [0; n\ -\ 1\ right\ ], p\ rSub\ \{\ size\ 8\{n, k\}\ \}\ =\ \{\ \{C\ rSub\ \{\ size\ 8\{n\ -\ 1\}\ \}\ rSup\ \{\ size\ 8\{k\}\ \}\ \}\ over\ \{2\ rSup\ \{\ size\ 8\{$

2. Evaluation de la complexité en moyenne du sous-algorithme `développe`

Soit $T(n)$ la complexité en moyenne de ce sous-algorithme lorsqu'on l'applique à une forme arborescente de taille n. Remarquons que $T(0)$ est la complexité de l'instruction `Feuille -> germe`.
Soit c la complexité de la reconnaissance de motif `|Tronc ... ->` et de l'instruction d'évaluation booléenne `if r1 < 0.5` . et d celle du retour de la fonction.
Calculons $T(n)$ pour $n^3 1$. On évalue d'abord la condition (complexité c), puis selon le résultat, on applique récursivement `développe` à la branche gauche (probabilité $\frac{1}{2}$) ou à la branche droite (probabilité $\frac{1}{2}$). Si on l'applique à la branche gauche, et que celle-ci est de taille k (probabilité $p_{n,k}$), on a une complexité égale à $T(k)$. Si on l'applique à la branche droite, et que la branche gauche est de taille k (probabilité $p_{n,k}$), alors la branche droite est de taille $n - 1 - k$, donc la complexité est $T(n - 1 - k)$. Enfin on retourne le nouveau tronc (complexité d). Donc, pour $n^3 1$: $T(n) = c + \sum_{k=0}^{n-1} \frac{1}{2} p_{n,k}(T(k) + T(n - 1 - k)) + d = c + d + \sum_{k=0}^{n-1} p_{n,k} T(k)$.
Soit $a^3 \frac{c+d}{ln2}$. Montrons par récurrence que $T(n) \leq aln(n + 1) + T(0)$.

- c'est manifestement vrai pour n = 0.

- si c'est vrai pour $0 \leq k \leq n - 1$, avec $n^3 1$, alors $T(n) \leq c + d + T(0) \sum_{k=0}^{n-1} p_{n,k} + a \sum_{k=0}^{n-1} p_{n,k} ln(k + 1)$.

Or $\sum_{k=0}^{n-1} p_{n,k} = 1$, car c'est une somme exhaustive de probabilités, ce qui permet de simplifier la première partie de l'expression ci-dessus, mais a aussi pour conséquence que l'on peut appliquer la formule de concavité généralisée à la fonction ln aux points 1, 2,..., n : $\sum_{k=0}^{n-1} p_{n,k} ln(k + 1) \leq ln\left(\sum_{k=0}^{n-1} p_{n,k}(k + 1)\right)$. On a ainsi :
$T(n) \leq c + d + T(0) + aln\left(\sum_{k=0}^{n-1} p_{n,k}(k + 1)\right) \leq aln2 + T(0) + aln\left(1 + \sum_{k=0}^{n-1} kp_{n,k}\right) = T(0) + aln2\left(1 + \sum_{k=0}^{n-1} kp_{n,k}\right)$
.
Un dénombrement classique fournit $\sum_{k=0}^{n-1} kC_{n-1}^{k} = (n - 1)2^{n-2}$, d'où
$aln2\left(1 + \sum_{k=0}^{n-1} kp_{n,k}\right) = aln2\left(1 + \frac{n-1}{2}\right) = aln(n + 1)$.
Enfin, on aboutit bien à $T(n) \leq aln(n + 1) + T(0)$. On en déduit que :
la complexité en moyenne de `développe` est en $O(\ln n)$ par rapport à la taille n de la forme arborescente

3. Evaluation de la complexité en moyenne de `construit_forme n`

Soit $U(n)$ cette complexité. Il est clair que $U(n) = O(1) + U(n - 1) + T(n - 1) = U(n - 1) + O(lnn)$.
On en déduit que $U(n) = U(0) + O(lnn!)$. La Formule de Stirling fournit un équivalent de $lnn!$: $lnn! \gg nlnn$.
On peut en conclure que $U(n) = O(nlnn)$.

Conclusion : la complexité en moyenne de `construit_forme1 n` est en $O(n\ln n)$

Remarque :
Il est évident que la complexité dans le pire des cas de `développe` (arbre peigne avec systématiquement `r1 > 0.5`) est linéaire. On en déduit que la complexité dans le pire des cas est quadratique.

Question 11

Dans cette question, on ne précise pas comment les éléments de la matrice de ramification sont pris en compte pour construire la forme arborescente (ce sera précisé à la question suivante). Seule la taille s de la matrice est exploitée.

On ne cherche donc plus à contrôler le nombre de noeuds de la forme, mais plutôt son épaisseur. Remarquons que dans ce contexte la fonction Caml : `function k -> 1 + (random__int k)` remplit parfaitement ce rôle...

Pour décrire l'algorithme récursif **construit_forme2** correspondant, nous allons utiliser la représentation d'une forme arborescente proposée en question 8, en convenant qu'une arête terminale d'étiquette *n* sera représentée par `Tronc (Feuille,{épaisseur = n ; longueur = 0.0 ; angle = 0.0},Feuille)`.
Nous utiliserons l'algorithme **épaisseur** de la question 9, et nous supposerons construit un sous-algorithme récursif, de type parcours préfixe, nommé **feuilleter**, consistant à retourner un arbre dans lequel toute occurrence de `Tronc (Feuille,{épaisseur = 1 ; longueur = 0.0 ; angle = 0.0},Feuille)` dans l'arbre argument a été remplacé par `Feuille`.

- Un sous-algorithme utile

  Comme dans **construit_forme1**, nous allons avoir besoin d'un sous-algorithme **ramifier** pour trouver, dans une forme arborescente donnée, une arête terminale d'épaisseur strictement supérieure à 1, et la ramifier. De plus, ce sous-algorithme devra tenir compte de la valeur d'un booléen *trouvé*, qui signale si une telle arête a déjà été ramifiée, pour empêcher qu'au cours d'un même appel de **construit_forme1** ne soient créés plus de deux nouveaux noeuds. Le booléen *trouvé* sera donc mis à VRAI dès qu'une arête est ramifiée. S'il est à FAUX à la fin de **ramifier**, cela signifie que l'arbre ne peut plus être ramifié.
  Enfin, pour que la recherche d'une arête terminale soit aléatoire, on se servira de la fonction `r1` de la question 10.

```
ALGO ramifier (a : forme_arborescente) → forme_arborescente
DEBUT
  SI trouvé OU épaisseur (a) = 1 ALORS a
  SINON
        à_gauche ← (r1 < 0.5)
        SI a = Tronc (Feuille,noeud,Feuille) ALORS
              n ← épaisseur (a)
              i ← r2_n
              trouvé ← VRAI
              SI i = n ALORS
                    Tronc (     Tronc(Feuille,{épaisseur=n - 1;longueur=0.0;angle
= 0.0},Feuille),
                                      noeud,
                                      Tronc(Feuille,{épaisseur=n -
1;longueur=0.0;angle = 0.0},Feuille))
              SINON
                    SI à_gauche ALORS
                          Tronc (
Tronc(Feuille,{épaisseur=i;longueur=0.0;angle = 0.0},Feuille),
                                            noeud,

Tronc(Feuille,{épaisseur=n;longueur=0.0;angle = 0.0},Feuille))
                    SINON
                          Tronc (
Tronc(Feuille,{épaisseur=n;longueur=0.0;angle = 0.0},Feuille),
                                            noeud,

Tronc(Feuille,{épaisseur=i;longueur=0.0;angle = 0.0},Feuille))
                    FIN SI
              FIN SI
        SINON
```

```
                    SI à_gauche ALORS
                        g ← ramifie (sous_arbre_gauche (a))
                        Tronc (g,noeud,ramifie (sous_arbre_droit (a))
                    SINON
                        d ← ramifie (sous_arbre_droit (a))
                        Tronc (ramifie (sous_arbre_gauche (a)),noeud,d)
                    FIN SI
                FIN SI
            FIN SI
        FIN
```

- L'algorithme principal

```
ALGO construit_arbre2 (a) → forme_arborescente
DEBUT
  trouvé ← FAUX
  r ← ramifie (a)
  SI trouvé ALORS construit_arbre2 (r)
  SINON feuilleter (r)
  FIN SI
FIN
```

- Il ne reste plus alors qu'à appeler :
  construit_arbre2 (`Tronc(Feuille,{épaisseur = s ;longueur=0.0 ; angle = 0.0},Feuille))`)

L'algorithme ne se termine que s'il est à un moment donné appelé sur un arbre dont toutes les arêtes terminales sont d'épaisseur 1. L'épaisseur initiale étant s, ceci suppose que les sous-arbres créés par **ramifie** soient d'épaisseur décroissante non constante. Or il n'y a décroissance que dans le cas où $r2_n = n$, ce qui n'a qu'une probabilité $0 < p \leq 1$ de se produire. Si $p < 1$, il est donc théoriquement possible (par exemple en construisant un arbre peigne) d'avoir indéfiniment une arête terminale d'étiquette s, ou d'une autre valeur comprise entre 2 et s.

On n'est pas assuré de la terminaison de cet algorithme

Remarquons néanmoins que la probabilité asymptotique de ne jamais avoir $r2_n = n$ pour un certain n est, lorsque q est le nombre d'appels récursifs de **construit_arbre2**, de $(1-p)^q q \vec{\to} ¥0$. On peut donc en conclure à la certitude de la terminaison de l'algorithme au sens probabiliste, bien que, naturellement, les limites de la machine ne permettent pas forcément d'attendre le bon vouloir de la Fortune...

Question 12

Une possible implémentation en Caml des fonctions $r2_k$ à partir de la matrice de ramification `M` est :

```
let r2 k M =
  let t = sub_vect M.(k) 0 k and r1 = random__float 1.0 and i = ref 0
  in
  for i = 1 to k - 1 do t.(k) <- t.(k) +. t.(k-1) done;
  while r1 >= t.(!i) do incr i done;
  !i + 1;;
```

On peut espérer avec cette méthode que les formes arborescentes construites auront, en moyenne, une matrice de ramification proche de celle qui a permis leur construction. D'après la Question 5, elle permettra donc de contrôler avec une bonne probabilité de réussite le caractère touffu ou effilé de la forme arborescente : plus la matrice de ramification utilisée sera remplie, et plus l'arbre sera touffu.

Cette technique permet d'espérer construire des formes arborescentes ayant un aspect plus ou moins effilé ou touffu

Deuxième Partie

Question 13

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

Ce programme utilise 4 registres

Question 14

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

Ce programme utilise 3 registres

Question 15

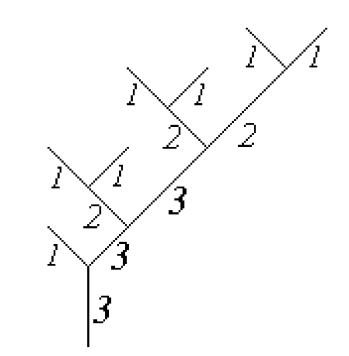
$R_0 \neg d$; $R_1 \neg e$; $R_0 \neg R_0 - R_1$; $R_1 \neg f$; $R_2 \neg g$; $R_1 \neg R_1 + R_2$; $R_0 \frac{\neg R_0}{R_1}$; $R_1 \neg b$; $R_2 \neg c$; $R_1 \neg R_1 - R_2$;
$R_0 \neg R_1 * R_0$; $R_1 \neg a$; $R_0 \neg R_1 + R_0$ (en utilisant l'associativité à droite pour le deuxième terme de la somme)

Ce programme utilise 3 registres

Question 16

$$+ \quad N\ O \quad a \quad N\ O \quad - \quad O \quad N\ O \quad b \quad c\ N\ O \quad N\ O \quad - \quad + \quad N\ O \quad N\ O \quad d \quad e \quad f$$

Question 17



Rappelons le théorème d'induction structurelle pour les expressions arithmétiques : si une proposition définie sur ces expressions est vraie pour toute expression réduite à un opérande (constante ou variable), et si, dès lors qu'elle est vraie pour deux expressions, elle est vraie pour toute expression formée à l'aide d'un opérateur quelconque appliqué à ces deux expressions, alors cette propriété est vraie pour toutes les expressions arithmétiques.
Montrons par induction structurelle la proposition suivante des expressions arithmétiques :
L'épaisseur de la forme arborescente associée à une expression arithmétique est égale au nombre de registres utilisés pour l'évaluation de cette dernière avec la stratégie optimale exposée à la Question 15

- C'est vrai si l'expression arithmétique est réduite à une constante ou une variable : en effet, dans ce cas, la forme associée est une feuille, dont l'épaisseur vaut 1, tandis qu'il ne faut qu'un seul registre pour charger une constante ou une variable :
  $R_0 \neg constante$ ou $R_0 \neg MEM[j]$ (cas d'une variable qui pointe vers l'emplacement-mémoire $MEM[j]$)
- Supposons que ce soit vrai pour les deux sous-expressions d'une expression arithmétique non réduite à un seul opérande, et qu'il faille g registres pour évaluer la sous-expression gauche et d registres pour la droite. Appliquons la stratégie optimale à cette expression. Deux cas peuvent se produire :

- ou bien l'une des deux sous-expressions requiert pour son évaluation un nombre de registres strictement supérieur à l'autre, et alors on les évalue dans cet ordre. Supposons pour fixer les idées que $g > d$.
Alors, par hypothèse d'induction, la forme arborescente associée à notre expression a une biépaisseur égale à (d,g), et donc une épaisseur égale à g.
Mais, par ailleurs, il faut g registres pour évaluer la sous-expression gauche, soit $R_0,..,R_{g-1}$, et le résultat est stocké dans R_0, puis il en faut d pour évaluer la sous-expression droite, soit $R_1,..,R_d$, et le résultat est stocké dans R_1 ; enfin, le résultat final de l'évaluation est stocké dans R_0. Comme $d \leq g - 1$, il aura donc bien fallu dans ce cas *g* registres pour l'ensemble de l'évaluation.

- ou bien *g* = *d*, et dans ce cas le même raisonnement montre que les d + 1 registres $R_0,..,R_d$ sont nécessaires à l'évaluation complète de l'expression, tandis que la biépaisseur de la forme associée étant maintenant (d,d), son épaisseur vaut également d + 1.

On peut donc conclure : <u>L'épaisseur de la forme arborescente associée à une expression arithmétique est égale au nombre de registres utilisés pour l'évaluation de cette dernière avec la stratégie optimale exposée à la Question 15</u>

Troisième Partie

<u>Remarque préalable</u>

L'algorithme de codage des formes arborescentes sans contrainte de degré 2 n'est qu'illustré par un exemple. Il est donc nécessaire d'expliciter d'abord cet algorithme. Cela peut être fait de façon récursive, en remarquant qu'une forme arborescente comporte toujours au moins une arête, et que toute branche d'une forme sans contrainte de degré 2 est elle-même une forme arborescente sans contrainte de degré 2 :

ALGO **codage** (f : *forme arborescente sans contrainte de degré* 2) → *mot de* S_E
DEBUT
CAS
SI f = ALORS a [cas de l'arbre réduit à une feuille]
SI f =  ALORS a**codage** (f') [concaténation du symbole a avec le codage du reste de la forme]
SI f =  OU  ALORS a[**codage** (f'')]**codage** (f')
FIN CAS
FIN

Question 18

Pour avoir le codage de 10(i) dans S_E, il suffit de remplacer toutes les lettres de ω par des a. On obtient le mot :

$$\underline{aa[aa]aa[a[a]a]a[aaa]aa}$$

Question 19

Un sous-mot $[w]$ d'un mot u associé à une forme arborescente n'est pas forcément le codage d'une branche de cette forme : par exemple, le sous-mot $[a[a]$ du mot de la Question 18 ne représente pas le codage d'une branche de 10(i). Attention, ce n'est pas seulement une question de non-égalité du nombre des crochets ouvrants et des crochets

fermants, comme le prouve l'exemple du sous-mot $[a[a]a]a[aaa]$. Ce qui est déterminant, c'est le fait que $a[a$et $a[a]a]a[aaa$ne sont pas bien emboîtés. On peut ainsi donner la caractérisation suivante :

Le sous-mot [ω] de u représente le codage d'une branche si et seulement si ω est bien emboîté

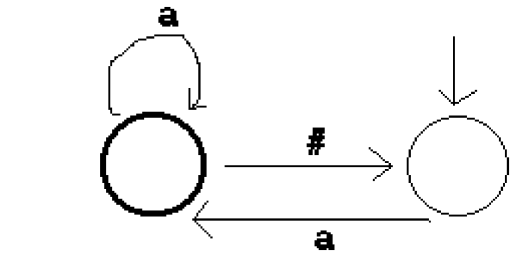
Question 20

Pour montrer ce résultat sur les branches d'une forme f, il suffit de montrer que toute forme peut s'écrire de façon unique $x_1[\alpha_1]\ldots[\alpha_n]x_{n+1}$, avec $x_k \hat{I} S_E^+$, les $[\alpha_k]$étant des branches. Nous allons procéder par induction structurelle, en reprenant les trois cas de l'algorithme **codage** détaillé plus haut :

- Si *f* est une feuille, alors **codage** (*f*) = a. L'existence et l'unicité sont donc vraies avec $n = 0$.
- Si *f* a une racine de degré 1, dont le reste de l'axe est *f'*, alors **codage** (*f*) = a**codage** (*f'*).
  Par hypothèse d'induction, **codage** (*f'*) s'écrit de façon unique $x_1[\alpha_1]\ldots[\alpha_n]x_{n+1}$, donc, en posant $x_1^{'} = ax_1$, on a bien **codage** (*f*) = $x_1^{'}[\alpha_1]\ldots[\alpha_n]x_{n+1}$.
- Si *f* a une racine de degré 2, alors on a **codage** (*f*) = a[**codage** (*f ''*)]**codage** (*f'*).
  Par hypothèse d'induction, on a de façon unique **codage** (*f'*) = $x_1[\alpha_1]\ldots[\alpha_n]x_{n+1}$. Posons $x_0 = a$et α_0 = **codage** (*f ''*). On a alors bien l'unique écriture **codage** (*f*) = $x_0[\alpha_0]x_1[\alpha_1]\ldots[\alpha_n]x_{n+1}$.

Question 21

Les mots de S qui peuvent être obtenus comme axes marqués de branches de codages de formes arborescentes sont tous ceux constitués d'un nombre quelconque (non nul) de symboles a, suivis par une nombre quelconque (éventuellement nul) de séquences de symboles formées par un # suivi d'un nombre quelconque (non nul) de symboles a. Ils correspondent donc à l'expression rationnelle $aa*$. Cette caractérisation permet de construire un automate reconnaissant le langage formé par ces mots :



Question 22

Les mots de S_E qui peuvent être considérés comme le codage d'arbres parfaits sont de la forme a ou $a\ ou\ a[w]w$, où ω est le codage d'un arbre parfait. On peut donc aisément construire un algorithme récursif permettant de tester si un mot de $S_E$est le codage d'un arbre parfait :

ALGO **est_parfait** (*mot*) → *booléen*
DEBUT
SI mot = « a » ALORS VRAI
SINON
SI *mot* commence par « a[» ALORS
$k \leftarrow 1$
$m \leftarrow$ *mot* privé de ses deux premiers symboles a et [
$m' \leftarrow \Lambda$
TANT QUE $k^1 0$ [*mot* étant bien emboîté, cette condition finira par être remplie avant que m soit vide]

BOUCLER

SI le 1er symbole de m est « [» ALORS $k \leftarrow k$ + 1 FIN SI

SI le 1er symbole de m est «] » ALORS $k \leftarrow k$ − 1 FIN SI

$m' \leftarrow m'$ concaténé à droite avec le 1er symbole de m

$m \leftarrow m$ privé de son 1er symbole

FIN BOUCLE

$m' \leftarrow m'$ privé de son dernier symbole [qui est nécessairement un «] »]

SI m = m' ALORS **est_parfait** (m) SINON FAUX FIN SI

SINON FAUX

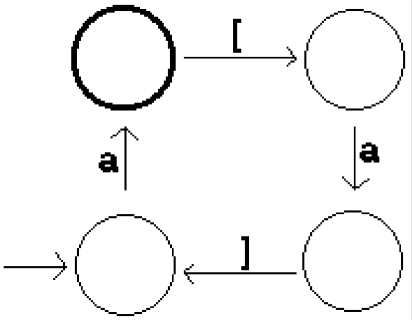
FIN SI

FIN SI

FIN

Question 23

Les mots de S_E qui peuvent être considérés comme le codage d'arbres peignes sont formés d'un symbole a suivi d'un nombre quelconque (éventuellement nul) de séquences « [a]a ». Ils correspondent donc à l'expression rationnelle $a([a]a)*$.
Voici un automate capable de reconnaître ces mots :



Question 24

Un mot de S_E est le codage d'une forme arborescente s'il est de la forme $x_1[\alpha_1]x_2[\alpha_2]\ldots x_n[\alpha_n]x_{n+1}$, où $n^3 0$, les x_k sont des mots de S^+, tandis que les α_k sont des codages de formes arborescentes. On est donc amené, pour tester si un mot de S_E est le codage d'une forme arborescente, à proposer l'algorithme récursif suivant :

ALGO **est_forme_arborescente** (mot) → *booléen*
DEBUT

$m \leftarrow mot$

SI $m = L$ ALORS FAUX

SINON

TANT QUE $m^1 L$ ET le 1er symbole de m est « a »

BOUCLER

$m \leftarrow m$ privé de son 1er symbole

FIN BOUCLE

SI $m = L$ ALORS VRAI

SINON

SI le 1er symbole de m est «] » ALORS FAUX

SINON [le 1er symbole de m est nécessairement « [»]

$m \leftarrow m$ privé de son 1er symbole

$m' \leftarrow \Lambda$

TANT QUE $m^1 L$ ET le 1er symbole de m est différent de «] »

BOUCLER

$m' \leftarrow m'$ concaténé à droite avec le 1er symbole de m

$m \leftarrow m$ privé de son 1er symbole
FIN BOUCLE
SI $m = L$ALORS FAUX
SINON
$m \leftarrow m$ privé de son 1er symbole [qui est un «] »]
est_forme_arborescente (m') ET **est_forme_arborescente** (m)
FIN SI
FIN SI
FIN SI
FIN SI
FIN

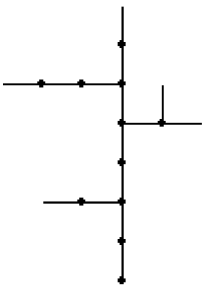
Question 25

Notons qu'en Caml, la fonction « tracer » correspond à `lineto`, tandis que « aller » correspond à `moveto` : nous écrirons donc l'algorithme demandé en Caml.
Remarquons d'autre part, en observant l'exemple de la Figure 10, que le mot censé donner un codage de la forme arborescente à tracer ne contient pas d'information quant au chemin de la racine à une feuille qui est choisi pour représenter l'axe de la forme arborescente. On peut donc imaginer de laisser ce choix au hasard, en se servant d'une fonction du type r_1, définie à la Question 10. A chaque survenue d'une branche latérale, on appelle cette fonction : si elle renvoie un nombre entre 0 et 0,5 on « fait pousser » une branche latérale vers la gauche, sinon on la fait pousser vers la droite.
Enfin, puisqu'on ne nous demande pas d'optimisation du dessin, et que les Figures 7 et 8 ne suggèrent pas un choix privilégié, nous allons choisir la simplicité : chaque segment aura la même longueur (1 « unité » = 20 pixels), et chaque branche latérale fera avec l'axe un angle droit :

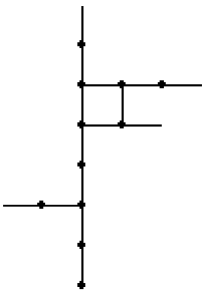
```
let rec trace_forme mot x0 y0 direction =
  if mot = "" then ()
  else
    match mot.[0] with
     `[` ->
        begin
        let k = ref 1 and compteur = ref 1 and hasard = random__float 1.0 < 0.5
        in
        while !compteur <> 0
        do if mot.[!k] = `]` then decr compteur else if mot.[!k] = `[` then incr compteur; incr k done;
        match (direction,hasard) with
           "haut",true -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "gauche";
                          moveto x0 y0;
                          trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "haut"
         |"haut",false -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "droite";
                          moveto x0 y0;
                          trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "haut"
         |"bas",true -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "droite";
                        moveto x0 y0;
                        trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "bas"
         |"bas",false -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "gauche";
                         moveto x0 y0;
                         trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "bas"
         |"gauche",true -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "bas";
                           moveto x0 y0;
                           trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "gauche"
         |"gauche",false -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "haut";
                            moveto x0 y0;
                            trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "gauche"
         |"droite",true -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "haut";
                           moveto x0 y0;
                           trace_forme (sub_string mot !k (string_length mot - !k)) x0 y0 "droite"
         |"droite",false -> trace_forme (sub_string mot 1 (!k - 2)) x0 y0 "bas";
                            moveto x0 y0;
                            trace_forme (sub_string mot !k (string_length mot - !k )) x0 y0 "droite"
        end
```

```
|`a` ->
  begin
  fill_circle x0 y0 2;
  match direction with
     "haut" -> lineto x0 (y0 + 20);
               trace_forme (sub_string mot 1 (string_length mot - 1)) x0 (y0 + 20) "haut"
    |"gauche" -> lineto (x0 + 20) y0;
                 trace_forme (sub_string mot 1 (string_length mot - 1)) (x0 + 20) y0 "gauche"
    |"droite" -> lineto (x0 - 20) y0;
                 trace_forme (sub_string mot 1 (string_length mot - 1)) (x0 - 20) y0 "droite"
    |"bas" -> lineto x0 (y0 - 20);
              trace_forme (sub_string mot 1 (string_length mot - 1)) x0 (y0 - 20) "bas"
  end;;
```

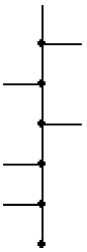
Et la commande suivante, exécutée deux fois consécutivement, donne les deux formes suivantes, équivalentes en ce qui concerne leur structure de branchement, et représentant l'arbre de la Figure 10 :



```
clear_graph ();;
moveto 400 300;;
trace_forme "aa[aa]aa[a[a]a]a[aaa]aa" 400 300 "haut";;
```



On a de même pour un arbre_peigne :



```
clear_graph ();;
moveto 400 300;;
trace_forme "a[a]a[a]a[a]a[a]a[a]a" 400 300 "haut";;
```