Correction du Devoir Surveillé du Samedi 11 Octobre

Exercice 1 (Arbres binaires et polynômes)

1. L'expression est $(5 + (7 \times 3)) + (0 \times 9) = 26$.
2.

```
let exple = N(Addition, N(Addition, F 5., N(Multiplication, F 7., F 3.)),
            N(Multiplication, F 0., F 9.)) ;;
```

3.

```
let rec evalue_exp e = match e with
  | F x -> x
  | N(Addition, eg, ed) -> (evalue_exp eg) +. (evalue_exp ed)
  | N(Multiplication, eg, ed) -> (evalue_exp eg) *. (evalue_exp ed)
;;
```

4. Cet arbre représente le polynôme $P(X) = (5 + 7X) \times 3X = 15X + 21X^2$. Sa valeur en 2 est $P(2) = 114$.
5. (a)

```
type polynome =
  | C of float
  | X
  | N of operation * polynome * polynome
;;
```

   (b)

```
let exple = N(Multiplication, N(Addition, C 5.,
      N(Multiplication, X, C 7.)), N(Multiplication, C 3., X)) ;;
```

6.

```
let rec evalue_poly p x = match p with
  | C c -> c
  | X -> x
  | N(Addition, pg, pd) -> (evalue_poly pg) +. (evalue_poly pd)
  | N(Multiplication, pg, pd) -> (evalue_poly pg) *. (evalue_poly pd)
;;
```

7. (a)

```
let rec add_poly p1 p2 = match (p1, p2) with
  | [], [] -> []
  | _, [] -> p1
  | [], _ -> p2
  | t1::q1, t2::q2 -> (t1 +. t2)::(add_poly q1 q2)
;;
```

   (b)

```
let rec mult_poly p1 p2 = match (p1, p2) with
  | _, [] -> []
  | [], _ -> []
  | t1::q1, t2::q2 -> let c1 = 0. :: (List.map (( *. ) t1) p2) in
                      let c2 = 0. :: (List.map (( *. ) t2) p1) in
```

```
                let c3 = 0. :: 0. :: (mult_poly q1 q2) in
                add_poly [t1 *. t2] (add_poly c1 (add_poly c2 c3))
;;
```

(c)

```
let rec coefficients p = match p with
  | C c -> [c]
  | X -> [0.; 1.]
  | N(Addition, pg, pd) -> add_poly (coefficients pg) (coefficients pd)
  | N(Multiplication, pg, pd) ->
                mult_poly (coefficients pg) (coefficients pd)
;;
```

Exercice 2 (Les arbres AVL)

1. On raisonne par induction structurelle pour montrer que

$$h(A) + 1 \leqslant s(A) \leqslant 2^{h(A)+1} - 1.$$

— Si A est l'arbre vide, $s(A) = 0$ et $h(A) = -1$ et le résultat annoncé est bien vérifié.
— Si $A = (F_g, x, F_d)$, supposons le résultat acquis pour F_g et F_d. On a :

$$s(A) = 1+s(F_g)+s(F_d) \geqslant 1+h(F_g)+1+h(F_d)+1 \geqslant 2+\max(h(F_g), h(F_d)) = 1+h(A)$$

et

$$s(A) = 1+s(F_g)+s(F_d) \leqslant 2^{h(F_g)+1}+2^{h(F_d)+1}-1 \leqslant 2\times 2^{\max(h(F_g),h(F_d))+1}-1 = 2^{h(A)+1}-1.$$

Avec la première inégalité, on a $h(A) \leqslant s(A) - 1$. Avec la seconde, $s(A) + 1 \leqslant 2^{h(A)+1}$ et donc, par croissance du logarithme, $\log_2(s(A) + 1) - 1 \leqslant h(A)$. Finalement,

$$\log_2(s(A) + 1) - 1 \leqslant h(A) \leqslant s(A) - 1.$$

2. (a) On a (en reprenant la preuve de la question précédente pour la troisième équivalence) :

$$\begin{aligned}
A \text{ est complet} &\Leftrightarrow h(A) = \log_2(s(A) + 1) - 1 \\
&\Leftrightarrow s(A) = 2^{h(A)+1} - 1 \\
&\Leftrightarrow \begin{cases} 1 + s(F_g) + s(F_d) = 2^{h(F_g)+1} + 2^{h(F_d)+1} - 1 \\ 2^{h(F_g)+1} + 2^{h(F_d)+1} - 1 = 2 \times 2^{\max(h(F_g),h(F_d))+1} - 1 \end{cases} \\
&\Leftrightarrow \begin{cases} s(F_g) = 2^{h(F_g)+1} - 1 \\ s(F_d) = 2^{h(F_d)+1} - 1 \\ h(F_g) = h(F_d) \end{cases} \\
&\Leftrightarrow \begin{cases} F_g \text{ est complet} \\ F_d \text{ est complet} \\ h(F_g) = h(F_d) \end{cases}
\end{aligned}$$

(b) Pour le sens direct, on raisonne par induction structurelle :
— Si A est l'arbre vide, A est complet et toutes ses feuilles sont à la même profondeur (car il n'y a pas de feuille...).

— Si $A = (F_g, x, F_d)$ est complet, alors F_g et F_d sont complets et de même hauteur (d'après la question précédente). Par hypothèse d'induction, toutes les feuilles de F_g sont à la même profondeur et toutes les feuilles de F_d sont à la même profondeur. Comme F_g et F_d sont de même hauteur, toutes les feuilles de F_g et de F_d sont à la même profondeur. Donc toutes les feuilles de $A = (F_g, x, F_d)$ sont à la même profondeur.

Pour le sens réciproque, soit A est un arbre binaire non vide (sinon il est complet) dont toutes les feuilles sont à la même profondeur. On a alors nécessairement :

$$s(A) = 1 + 2 + 2^2 + \ldots + 2^{h(A)} = 2^{h(A)+1} - 1$$

et donc

$$h(A) = \log_2(s(A) + 1) - 1.$$

Ainsi, A est complet.

3. (a) Considérons une rotation droite et notons a, b et c des éléments respectifs des ABR A, B et C. Par hypothèse, $a \leqslant x < b$ et $b \leqslant y < c$.

Ces inégalités peuvent s'écrire de manière équivalente : $b \leqslant y < c$ et $a \leqslant x < b$, ce qui traduit que l'arbre obtenu par rotation droite est toujours un ABR.

Il en est bien sur de même d'une rotation gauche.

(b) On utilisera le type :

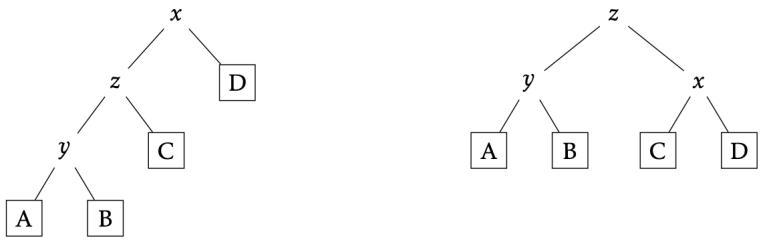
```
type arbre = V | N of arbre * int * arbre ;;
```

Les deux fonctions de rotations se réalisent à coût constant en écrivant :

```
let rotation_d ab = match ab with
  | N (N (a, x, b), y, c) -> N (a, x, N (b, y, c))
  | _ -> failwith "rotation_d"
;;
```

```
let rotation_g ab = match ab with
  | N (a, x, N (b, y, c)) -> N (N (a, x, b), y, c)
  | _ -> failwith "rotation_g"
;;
```

(c) La rotation gauche autour de y conduit à l'arbre de gauche. Avec la rotation droite autour de x, on obtient finalement l'arbre de droite.



4. (a) Nous allons montrer par induction structurelle que tout arbre AVL A de hauteur h contient au moins f_h sommets.
— Si A est l'arbre vide, alors $s(A) = 0 = f_0$.
— Si $A = (F_g, x, F_d)$, l'un des deux sous-arbres F_g ou F_d est de hauteur $h-1$ donc contient au moins f_{h-1} sommets, l'autre est au moins de hauteur $h-2$ donc contient au moins f_{h-2} sommets. On en déduit que A contient au moins $f_{h-1} + f_{h-2} + 1 = f_h + 1$ sommets.

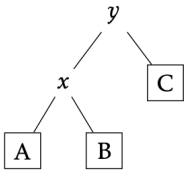
(b) D'après la question précédente, $s(A) \geqslant f_{h(A)}$.

Sachant que $f_h = O(\varphi^h)$ avec $\varphi = \dfrac{1+\sqrt{5}}{2}$, on en déduit :

$$\varphi^{h(A)} = O(s(A)) \quad \text{et donc} \quad h(A) = O(\log_2(s(A))).$$

5. (a) Le déséquilibre initial de y est égal à -1, 0 ou 1, et l'apparition d'une feuille parmi ses descendants ne peut modifier son déséquilibre que d'au plus une unité, donc $\text{eq}(y) = \pm 2$.

(b) Si on suppose $\text{eq}(y) = 2$, la situation peut être représentée ci-dessous :

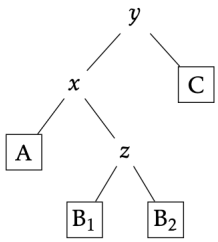


avec $h(C) = \max(h(A), h(B)) - 1$.

Si $\text{eq}(x) = 0$, alors $h(A) = h(B)$ et $h(C) = h(B) - 1$, et la rotation droite autour de y conduit aux nouvelles valeurs du déséquilibre $\text{eq}(x) = h(A) - (h(B) + 1) = -1$ et $\text{eq}(y) = h(B) - h(C) = 1$ donc la condition AVL est de nouveau respectée.

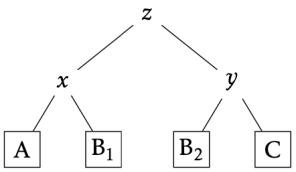
Si $\text{eq}(x) = 1$, alors $h(A) = h(B) + 1$ et $h(C) = h(B)$, et la rotation droite autour de y conduit aux nouvelles valeurs du déséquilibre $\text{eq}(x) = h(A) - (h(B) + 1) = 0$ et $\text{eq}(y) = h(B) - h(C) = 0$ donc la condition AVL est de nouveau respectée.

(c) Si $\text{eq}(x) = -1$, alors $h(B) = h(A) + 1$ et $h(C) = h(A)$. Puisque $h(B) \geqslant 0$, B n'est pas l'arbre vide. Notons z sa racine, B_1 et B_2 ses fils gauche et droit. La situation est alors la suivante :



avec $\max(h(B_1), h(B_2)) = h(A) = h(C)$. Comme $\text{eq}(z) = -1, 0$ ou 1 (car y est le premier ancêtre qui ne respecte pas la condition AVL), on a $h(B_1) - h(B_2) = -1, 0$ ou 1.

La question 3.(c) a montré qu'en deux rotations il était possible d'obtenir l'arbre ci-dessous :



Les nouvelles valeurs du déséquilibre sont $\text{eq}(x) = 0$ ou 1, $\text{eq}(y) = 0$ ou -1, $\text{eq}(z) = 0$. Donc la condition AVL est de nouveau respectée.

Exercice 3 (Algorithmique des mots sans facteur carré)

1. Voici le programme demandé :

```
let rec longueur l = match l with
  | [] -> 0
  | t::q -> 1 + longueur q
;;
```

2. Voici le programme demandé (on renvoie la sous-liste de l de longueur n débutant à l'indice k ; si n est trop grand, renvoie la sous-liste débutant à l'indice k et finissant au bout de la liste) :

```
let rec sous_liste l k n =
  if n = 0 then []
  else
    match l with
      | [] -> []
      | t::q when k=0 -> t::(sous_liste q 0 (n-1)
      | t::q -> sous_liste q (k-1) n
;;
```

3. (a) $aabfa$ contient le facteur carré aa.
   (b) La seule lettre qui se répète est le a, mais les deux a ne sont pas côte à côte et ne forment donc pas un facteur carré.
   (c) $abab$ est un facteur carré dans $ababa$.
   (d) Même raisonnement que pour la (b) : pas de facteur carré.
4. Notons a la première lettre de w. Si w contient seulement des a c'est évident. Sinon, soit b l'autre lettre utilisée. Si w contient aa ou bb c'est réglé. Sinon, les seuls facteurs le longueur 2 possibles sont ab et ba. Alors w commence par $abab$, qui est un facteur carré.
5. Voici le programme demandé :

```
let estCarre l =
  let n = longueur l in
    if n mod 2 <> 0 then false
    else
      let u = sous_liste l 0 (n/2) and v = sous_liste l (n/2) (n/2) in
        u = v
;;
```

6. Soit l une liste et n sa longueur. Les appels à `longueur` et à `sous_liste` n'effectuent pas de comparaisons de lettres et donc ne sont pas comptabilisés ici. Reste le `u = v` final, qui nécessite $n/2$ comparaisons de lettres. D'où une complexité en $O(n)$.
7. Voici le programme demandé :

```
let rec contientRepetitionAux l m = match l with
  | [] -> false
  | t::q -> if estCarre (sous_liste l 0 (2*m)) then true
            else contientRepetitionAux q m
;;
```

8. Soit u une répétition dans w. Vu les définitions, cela signifie qu'il existe trois mots p, x et s tel que $w = pxxs$. Alors :

$$n = |w| = |p| + 2|x| + |s| \quad \text{d'où} \quad |x| \leqslant \frac{n}{2}.$$

9. Voici le programme demandé :

```
let rec contientRepetition l =
  let n = longueur l in
    let rec boucle m =
      if m = 0 then false
      else (contientRepetitionAux l m) || (boucle (m-1))
    in
      boucle (n/2)
;;
```

10. Soit l une liste et n sa longueur.
    — Soit m un entier. L'exécution de `contientRepetitionAux l m` appelle `estCarre` sur chaque préfixe de l de taille $2m$, ce qui coûte $O(nm)$ comparaisons de lettres.
    — Lorsqu'on exécute `contientRepetition l`, la boucle appelle la fonction auxiliaire pour tout m entre 0 et $\lfloor n/2 \rfloor$, ce qui coûte

$$\sum_{m=0}^{\lfloor n/2 \rfloor} O(nm) = O\left(n \sum_{m=0}^{\lfloor n/2 \rfloor} m\right) = O(n^3).$$

11. *ababa*

12. • Supposons que uv contient un carré centré sur u. Reprenons les notations de la définition, de sorte que

$$\underbrace{u'(w'w'')(w'}_{u}\ \underbrace{w'')v'}_{u}$$

    Posons $i = |u'w'|$.
    Le mot w' est un suffixe commun à $u[0, i-1]$ et à u, donc $|lcs(u[0, i-1], u)| \geqslant |w'|$.
    De même, w'' est un préfixe commun à $u[i, :]$ (notation Python...) et à v, donc $|lcp(u[i, |u|-1, v)| \geqslant |w''|$.
    Au total,

$$|lcs(u[0, i-1], u)| + |lcp(u[i, |u|-1, v)| \geqslant |w'| + |w''| = |u| - |u'| - |w'| = |u| - i.$$

    • Réciproquement, supposons qu'il existe $i \in [\![0, |u|-1]\!]$ tel que

$$|lcs(u[0, i-1], u)| + |lcp(u[i, |u|-1, v)| \geqslant |w'| + |w''| = |u| - i.$$

    Fixons un tel i. Notons alors w' le plus long suffixe commun de $u[0, i-1]$ et u, et u' le "reste" de $u[0, i-1]$ (donc $u[0, i-1] = u'w'$).
    Notre hypothèse est que $|lcp(u[i, |u|-1, v)| \geqslant |u| - |w'| - i$. Prenons alors pour w'' un préfixe commun à $u[i, :]$ et v de longueur égale à $|u| - |w'| - i$. Et notons v' le reste de v.
    Maintenant, il faut remarquer que le mot $u[i, :]$ admet w'' comme préfixe, et w' comme suffixe, et que $|u[i, :]| = |w'| + |w''|$.
    Par conséquent, $u[i, :] = w''w'$. Il vient ensuite $u = u'w'w''w'$ et, comme $v = w''v'$, uv a bien un facteur carré centré en u.

13. $\text{pref}_u = [\![6; 1; 0; 0; 0; 1]\!]$, $\text{pref}_{u,v} = [\![1; 3; 0; 0; 0; 1]\!]$.

14. Voici le tableau demandé (en poussant l'algo un cran plus loin) :

i	f	g	pref[i]
0	–	0	12
1	1	3	2
2	2	3	1
3	3	3	0
4	4	12	8
5	4	12	2
6	4	12	1
7	4	12	0
8	4	12	4
9	4	12	2
10	4	12	1
11	4	12	0
12	12	12	0

15. Pour tout mot m, on note m^T l'image miroir de m.
Soit $n = |u|$. On a pour tout $i \in [\![0, n-1]\!]$, $(u^T).[i] = u.[n-1-i]$.
Soit $m \in \Sigma^*$. C'est un suffixe commun de u et de $u.[:i]$ si et seulement si m^T est un préfixe commun de u^T et de $(u^T).[n-i:]$. Par conséquent, $\text{suff}[i] = \text{pref}_{u^T}.[n-i]$.
D'où l'algorithme :

Entrées : Un mot u
Sorties : La table suff_u
$n \leftarrow |u|$
`pref_u_miroir` ← résultat de l'algo 1 appliqué à u^T
`suff` ← nouveau tableau de longueur $n+1$ initialisé à -1
pour *i de 0 à n* **:**
  `suff[i]` ← `pref_u_miroir[n-i]`
fin
Renvoyer `suff`

16. On appelle tabsuff1 la fonction décrite par l'algorithme de l'énoncé. On applique la méthode décrite en partie 4 :

Entrées : Deux mots u et v
Sorties : Le booléen « uv contient un facteur carré centré sur u»
`pref_uv` ← `tabpref(u,v)`
`suff_u` ← `tabsuff1(u)`
`res` ← Faux
$n \leftarrow |u|$
pour *i de 0 à* $n-1$ **:**
  `res` ← `res` ou `suff_u[i] + pref_uv[i]` $\geqslant n-i$
fin
Renvoyer Vrai

17. Le calcul des deux tableaux $\text{pref}_{u,v}$ et suff_u se fait en $O(|u|)$. La boucle coûte aussi $O(|u|)$. L'algo en entier est donc en $O(|u|)$.

18. On appelle `contient_carre` cette fonction. On appelle `contient_carre_centre_sur_u` la fonction de la question précédente et on suppose connue la fonction analogue `contient_carre_centre_sur_v`.

Entrées : Une chaîne de caractères m
Sorties : Le booléen «m admet un facteur carré».

si $|m| \leqslant 1$ **:**
  Renvoyer vrai
sinon :
  Soient u et v tels que $m = uv$ et $\big||u| - |v|\big| \leqslant 1$
  Renvoyer `contient_carré(u)` ou `contient_carré(v)` ou `contient_carré_centré_sur_u(u,v)` ou `contient_carré_centré_sur_v(u,v)`
fin

19. Pour tout $n \in \mathbb{N}$, soit C_n le nombre maximal de comparaisons de chaînes de caractères lors du calcul de `contient_carre` appliqué à une chaîne de longueur au plus n. Ainsi, $C_0 = 0$, $C_1 = 0$ et pour tout $n \geqslant 2$:

$$C_{\lfloor n/2 \rfloor} + C_{\lfloor (n+1)/2 \rfloor} + O(n),$$

les deux premiers termes venant des appels récursifs et le $O(n)$ de `contient_carre_centre_sur_u (u,v)` et `contient_carre_centre_sur_v (u,v)`.
Avec le théorème de complexité des DPR, on a donc $C_n = O(n \ln(n))$.