Corrigé de
Programmation fonctionnelle

1 Bases

1. `type t1 = X;;`
   `type t2 = X | Y of float;;`
   `type 'a t3 = A | B of int * 'a t3;;`
   `type t4 = F of t1 t3 | N of t4 t2;;`
   - `t1` est un type somme dégénéré. Il est équivalent au type `unit` (une seule valeur).
   - `t2` est un type somme à deux cas (constructeurs), soit une constante, soit un flottant.
   - `t3` est un type somme dont le second cas est un type produit. Il est polymorphe (`'a`) et récursif. On note que le paramètre de type est inutilisé (dont inutile). `t3` est équivalent à `int list`.
   - La définition de `t4` n'est pas syntaxiquement correcte car `t2` n'a pas de paramètre.

2.
```
#let p = fun n m f x -> n (m f) x;;

#let d = fun g y -> g (g y);;

#let q = p d d;;
```

   (a) Le type :

```
#q;;
- : ('_a -> '_a) -> '_a -> '_a = <fun>
```

   et `q` vaut `fun f x -> f (f (f (f x)))`, c-à-d l'entier de Church 4 (`p` est le produit, `d` est la représentation de 2).

2 Programmation

1. (a) Pour savoir si n est parfait, il s'agit de passer en revue tous les entiers inférieurs à n et sommer ceux qui le divisent. Par exemple en utilisant le récurseur (du à Gœdel) :

```
#let rec goedel = fun f n z ->
#  if n = 0
#  then z
#  else f (n-1) (goedel f (n-1) z);;
val goedel : (int -> 'a -> 'a) -> int -> 'a -> 'a = <fun>

#let parfait = fun n ->
#  let somme=goedel (fun k r -> if k >= 1 && n mod k = 0 then k + r else r) n 0 in
#  somme = n;;
val parfait : int -> bool = <fun>
```

   (b) Il s'agit encore d'une itération sur des entiers. Avec le récurseur :

```
#let enumere = fun n ->
#  goedel (fun k _ -> if parfait k then (print_int k; print_newline ())) (n+1) ();;
val enumere : int -> unit = <fun>
```

2. Il faut parcourir les deux listes en parallèle en supprimant le premier élément de la première quand on le trouve dans la seconde :

```
#let rec sous_liste = fun l1 l2 ->
#  match (l1, l2) with
#    [], _ -> true
#  | x::xs, y::ys -> if x = y then sous_liste xs ys else sous_liste l1 ys
#  | _ -> false;;
val sous_liste : 'a list -> 'a list -> bool = <fun>
```

3. Il est pratique d'écrire une fonction auxiliaire qui teste si une liste est préfixe d'une autre. Ensuite il suffit de tester si la première liste est préfixe de la seconde ou de sa queue.

```
#let rec prefixe = fun l1 l2 ->
#  match (l1,l2) with
#    [], _ -> true
#  | x::xs, y::ys -> x = y && prefixe xs ys
#  | _ -> false;;
val prefixe : 'a list -> 'a list -> bool = <fun>
```

```
#let rec sous_sequence = fun l1 l2 ->
#  prefixe l1 l2 ||
#  match l2 with
#    [] -> false
#  | _x::xs -> sous_sequence l1 xs;;
val sous_sequence : 'a list -> 'a list -> bool = <fun>
```

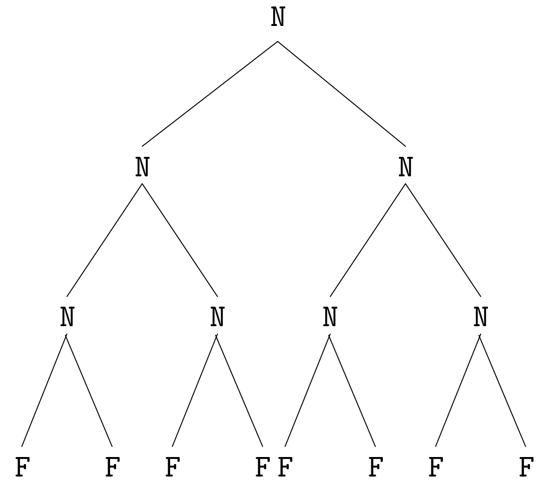
4.
```
#type b = F | N of b * b;;
```

```
#let rec f1 = fun n ->
#  if n = 0 then F else N (f1 (n-1), f1 (n-1));;
```

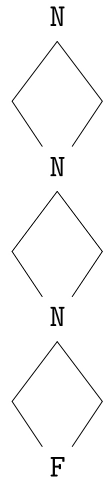
```
#let rec f2 = fun n ->
#  if n = 0 then F else let fn1 = f2 (n-1) in N (fn1, fn1);;
```

(a) – `f1` et `f2` produisent le même résultat (sémantique du `let`) ;
 – le temps de calcul de `(f1 n)` varie comme l'exponentielle de `n` ; le temps de calcul de `(f2 n)` varie linéairement avec `n`
 – les complexité spatiales des deux fonctions sont égales à leur complexité temporelle.

(b) Les sous-arbre construits par `f2` sont partagés à tous les niveaux :



`f1 3` : arbre complet



`f2 3` : partage

3 Mobile

1. On choisir de représenter pour chaque tige, sa longueur et la position (normalisée entre 0 et 1) de l'attache. Deux types : `t` pour une boule (avec sa masse) ou une tige et `mobile` pour un fil et un objet.

```
#type masse = float;;
```

```
#type longueur = float;;
```

```
#type t =
#   Tige of longueur*float*mobile*mobile (* (longueur,attache,gauche,droite) *)
#  | Boule of masse                      (* masse *)
#and mobile = float*t;;                  (* (longueur file, objet) *)
#let m = (2.,Tige (6., 0.41, (2.,Tige (6.,0.66,(1.,Boule 1.),(1.,Boule 2.))), (2.,Tige (2.,0.5,(1.,Boule 1.),(1.,Boule 1.)))));;
```

2. Récursion simple à travers l'arbre :

```
#let rec hauteur = fun m ->
#  match m with
#    (h, Boule _) -> h
#  | (h, Tige (_,_,m1,m2)) -> h +. max (hauteur m1) (hauteur m2);;
val m : float * t =
  2,
  Tige
   (6, 0.41, (2, Tige (6, 0.66, (1, Boule 1), (1, Boule 2))),
    (2, Tige (2, 0.5, (1, Boule 1), (1, Boule 1))))

#hauteur m;;
val hauteur : mobile -> float = <fun>
```

3. La fonction récursive vérifie l'équilibrage des sous-mobiles **et** calcule la masse. En cas d'échec, on abandonne avec une exception :

```
#let equilibre = fun m epsilon ->
# let zero = fun x -> x < epsilon in
# let rec equil = fun m ->
#  match m with
#    (_, Boule m) -> m
#  | (_, Tige (_, r, m1, m2)) ->
#     let m1 = equil m1 and m2 = equil m2 in
#     if not (zero (abs_float (r*.m1-.(1.-.r)*.m2))) then raise Exit;
#     m1 +. m2 in
#  try
#    let masse_equilibree = equil m in true
#  with
#    Exit -> false;;
- : float = 5

#equilibre m 0.01;;
val equilibre : mobile -> float -> bool = <fun>

#equilibre m 0.1;;
- : bool = false
```

4. Pour teste l'équivalence, le plus simple est de calculer une *forme normale* pour chaque arbre. On peut par exemple choisir de :
   (a) avoir un point d'attache pour chaque tige à gauche du milieu de la tige ;
   (b) en cas de centrage, mettre le mobile le plus lourd à gauche ;
   (c) en cas de centrage et d'égalité des masses, mettre le mobile le plus haut à gauche ;
   (d) en cas d'égalité des 3 critères précédents, mettre le mobile le plis large à gauche ;
   (e) ...