université de BORDEAUX	**Année Universitaire 2016/2017, Session 1 d'automne.** Licence Informatique L2, UE 4TINA01U, Programmation fonctionnelle. Jeudi 15 décembre 2016. 11h30–13h Durée: 1h30. **Tous documents autorisés.**	**Collège Sciences et Technologies**

Important.

- Seule la syntaxe vue en cours intégré et TD est autorisée. Vous pouvez utiliser les fonctions du module `List`.
- La notation attachera une grande importance à la clarté et à la concision des programmes et de leur justification. En particulier, les fonctions doivent être **commentées**.
- Les exercices sont indépendants. N'importe quelle question peut être faite en supposant les autres résolues.
- Le barème est indicatif.

Exercice 1 — Fonction break (4 points). Soit la fonction `break` suivante :

```
let rec break p l =
  match l with
  | [] -> ([], [])
  | a :: r -> let (l1, l2) = break p r
    in if p a then (a::l1, l2) else (l1, a::l2)
```

1) Quel est le type de la fonction `break` ?

2) Quelle est la valeur, en général, de `break p l` ? Justifiez votre affirmation par une preuve par récurrence.

3) Écrivez une version récursive terminale `breakfast` de la fonction `break`. Si vous utilisez une fonction auxiliaire, expliquez ce qu'elle calcule en fonction de ses arguments. ■

Solution.

1) La fonction `break` est du type `('a -> bool) -> 'a list -> 'a list * 'a list`. Le fait que le second argument est une liste se détecte grâce à la construction `match l with...` qui utilise comme motif une liste. Le type des éléments de la liste est quelconque, le second argument, `l`, est donc de type `'a list`. Le fait que la fonction retourne un couple de listes contenant des arguments de même type que la liste d'origine se détecte par la valeur retournée après le `then` et le `else`. Enfin, `p a` est utilisé comme condition, donc `p` est une fonction qui renvoie un Booléen. Elle s'applique sur un élément de la liste, elle est donc du type `'a -> bool`.

2) L'appel `break p l` retourne un couple `(l1,l2)` de deux listes où :

   - `l1` est la liste des éléments qui satisfont le prédicat `p`, dans le même ordre qu'ils apparaissent dans `l`,
   - `l2` est la liste des éléments qui ne satisfont pas le prédicat `p`, dans le même ordre qu'ils apparaissent dans `l`.

   On prouve cette propriété par récurrence sur la longueur de l'argument `l`. Si `l` est la liste vide, la fonction retourne `[],[]`, la propriété est donc vraie. Sinon, `l` est de la forme `a::r` où `r` est une liste de longueur strictement plus petite que `l`. On peut donc lui appliquer l'hypothèse de récurrence, et vérifier qu'elle se transmet à `l` en distingant le cas où `a` vérifie le prédicat `p` et celui où `a` ne le vérifie pas.

3)
```
let breakfast p l =
  let rec helper l acc1 acc2 =
    match l with
      [] -> List.rev acc1,List.rev acc2
    | a::r -> if p a then helper r (a::acc1) acc2 else helper r acc1 (a::acc2)
  in helper l [] []
```

Exercice 2 — Album photo (9 points). On veut manipuler une liste de descriptions de photos. Chaque description comporte une année de prise de vue et une liste de sujets. On définit donc les types suivants :

```
type sujet = Selfie | Monument | Miroir_d_Eau | Mode | People | Mon_assiette_au_resto
type annee = int
type photo = Photo of annee * (sujet list)
type album = photo list
```

Un exemple d'album est le suivant :

```
let mon_album = [ Photo(2016, [Selfie; Miroir_d_Eau]);
                  Photo(2014, [Selfie; People]);
                  Photo(2014, [Selfie; Monument; Mode]);
                  Photo(2012, [Mon_assiette_au_resto; People]) ]
```

1) Écrire une fonction `select_simple : sujet -> album -> album` telle que `select_simple s a` retourne la liste des photos de l'album `a` dont ***au moins un des sujets*** est `s`. Ainsi, `select_simple People mon_album` retourne `[Photo(2014, [Selfie; People]); Photo(2012, [Mon_assiette_au_resto; People])]`.

2) Écrire une fonction `select_by_date : (annee -> bool) -> album -> album` telle que `select_by_date p a` retourne la liste des photos de l'album `a` dont la date satisfait la fonction booléenne `p`. Par exemple, `select_by_date (fun x -> x >= 2014) a` doit renvoyer les liste des photos de `a` datées de 2014 ou après.

On définit maintenant le type `critere` suivant.

```
type critere =
  Sujet of sujet
| Date  of (annee -> bool)
| Ou    of critere * critere
| Et    of critere * critere
| Non   of critere
```

3) Écrire le critère spécifiant « le sujet contient `Selfie` mais pas `People` et la photo a été prise en 2014 ou après ».

4) Écrire une fonction `satisfait : critere -> photo -> bool` qui teste si une photo satisfait un critère.

5) Écrire une fonction `select : critere -> album -> album` telle que `select c a` renvoie la liste des photos de l'album `a` satisfaisant le critère `c`. ∎

Solution.

1)
```
let rec select_simple = fun sujet album ->
    match album with
        [] -> []
      | Photo(x,l)::q -> if List.mem sujet l then
                           Photo(x,l)::(select_simple sujet q)
                         else
                           select_simple sujet q
```

2)
```
let rec select_by_date = fun p album ->
    match album with
       [] -> []
     | Photo(x,l)::q -> if p x then
                          Photo(x,l)::(select_by_date p q)
                        else
                          select_by_date p q
```

3)
```
let selfie_recent a = Et (Et (Sujet Selfie, Non (Sujet People)), Date (fun d -> d >= 2014))
```

4)
```
let satisfait = fun c ph ->
    let Photo(a, l) = ph in
    let rec aux = fun c' ->
      match c' with
        Sujet s -> List.mem s l
      | Date p -> p a
      | Ou(c1,c2) -> aux c1 || aux c2
      | Et(c1,c2) -> aux c1 && aux c2
      | Non c1 -> not (aux c1)
    in aux c
```

5)
```
let rec select = fun c a ->
    match a with
       [] -> []
     | t::q -> if satisfait c t then
                 t::select c q
               else
                 select c q
```

Exercice 3 — Chemins dans les arbres (4 points). On utilise le type suivant pour les arbres binaires.

```
type 'a bintree = Leaf of 'a | Node of 'a bintree * 'a bintree
```

On représente un chemin dans un arbre par une suite de directions.

```
type direction = Left | Right
type path      = direction list
```

On appelle *branche* d'un arbre un chemin allant de la racine de l'arbre à une de ses feuilles.

1) Écrire une fonction `is_branch : path -> 'a bintree -> bool` telle que `is_branch p t` renvoie `true` si le chemin `p` est une branche de l'arbre `t` et `false` sinon.

2) Écrire une fonction `to_zero : int bintree -> path list` telle que `to_zero t` renvoie la liste de toutes les branches de `t` menant à une feuille étiquetée par 0. ∎

Solution.

```
    let son direction bintree =
  match bintree with
    Node(l, r) -> if direction = Left then l else r
  | _ -> failwith "unapplicable to a Leaf"

let rec is_branch path bintree =
  match bintree with
    Leaf _ -> path = []
  | Node(l, r) ->
     path != [] && is_branch (List.tl path) (son (List.hd path) bintree)

let add_direction direction paths =
  List.map (fun path -> direction :: path) paths

let rec to_zero bintree =
  match bintree with
    Leaf i -> if i = 0 then [[]] else []
  | Node(l, r) ->
     (add_direction Left (to_zero l)) @ (add_direction Right (to_zero r))
```

Exercice 4 — Hydres (3 points). Dans cet exercice, on utilise le type `hydra` suivant :

```
type hydra = Node of hydra list
```

L'arité d'un nœud d'une hydre est son nombre de filles.

Écrire une fonction `arite_max : hydra -> int` qui calcule l'arité maximum des noeuds de l'hydre. ■

Solution.

```
let rec arity_max hydra =
  match hydra with
    Node [] -> 0
  | Node hydras -> max (List.length hydras)
                       (List.fold_left max 0 (List.map arity_max hydras))
```