Feuille d'exercices n°3

Exercice I :

Q1 – Définir la fonction `repeat : int -> 'a -> ('a list)` telle que `(repeat n x)` donne la liste contenant `n` fois `x`.
Exemples:
`(repeat 7 true)` donne `[true; true; true; true; true; true; true]`
`(repeat 0 "Hello")` donne `[]`
`(repeat (-42) [1;2;3])` donne `[]`

Q2 – Définir la fonction `range : int -> int -> (int list)` telle que `(range i j)` donne la liste `[i; i+1; ..; j]`.
Remarque: si `i` $>$ `j` alors `(range i j)` donne la liste vide.

Q3 – Variante de la fonction précédente: définir la fonction `range_bis : int -> int -> (int list)` telle que `(range_bis x n)` donne la liste `[x; x+1; ..:x+n-1]`. Si `n` ≤ 0 alors `(range x n)` donne la liste vide.

Exercice II :

Q1 – Définir la fonction `begaie : ('a list) -> ('a list)` qui double la taille d'une liste en dupliquant chacun de ses éléments. Exemples:
`(begaie [])` donne `[]`
`(begaie [1;2;3])` donne `[1;1;2;2;3;3]`

Q2 – Définir la fonction `somme : (int list) -> int` qui donne la somme des éléments de la liste passée en argument.
Remarque: `(somme [])` donne `0`.

Q3 – Donnez une définition récursive terminale de `somme`.

Q4 – Définir la fonction `flatten : (('a list) list) -> ('a list)` qui concatène toutes les listes de son argument. Schématiquement, `(flatten [ [x1; ...; xn]; ..; [y1; ..; ym] ])` donne la liste `[x1; ...; xn]@ .. @[y1; ..; ym]`, c-à-d, `[x1; ..; xn; ..; y1; ..;ym]`.

Exemples:
`(flatten [])` donne `[]`
`(flatten [[]])` donne `[]`
`(flatten [[1;2;3]; []; [4;5;6;7]])` donne `[1;2;3;4;5;6;7]`

Exercice III :

Q1 – Définir la fonction `drop : int -> ('a list) -> ('a list)` telle que `(drop n xs)` donne la liste obtenue en privant `xs` de ses `n` premiers éléments. Si `n` ≤ 0, le résultat est `xs`; si `n` est plus grand que la taille de `xs`, le résultat est la liste vide.

Q2 – Définir la fonction `take : int -> ('a list) -> ('a list)` telle que `(take n xs)` donne la liste des `n` premiers éléments de `xs`. Que faire si `n` est négatif ou plus gand que la taille de la liste ?

Q3 – Déduire de ce qui précède la définition de la fonction `sub : ('a list) -> int -> int -> ('a list)` telle que `(sub start len xs)` extrait la sous liste de `xs` de longeur `len` qui commence à la position `start`. On aura que `(sub 0 (List.length xs) xs)`=`xs`.

Exercice IV : Listes d'association

Une liste d'association est un ensemble d'éléments associant des clés à des valeurs – concrètement, une liste de couples `(k,v)` où `k` est la clé et `v` la valeur associée. Nous allons définir la fonction `list_assoc` qui permet, à partir d'une clé et d'une liste d'association, de retrouver sa valeur associée. L'exception prédéfinie `Not_found` est déclenchée si la clé n'est pas présente.

Exemple
```
let dico = [ (1, "foo") ; (23, "baz"); (6, "bar") ];;
# list_assoc 6 dico;;
- :  string = "bar"
# list_assoc 3 dico;;
Exception :  Not_found.
```

Q1 – Donner le type de la fonction `list_assoc`.

Q2 – Définissez la fonction `list_assoc`. Si la clé existe deux fois, on donnera la première occurence trouvée.

Q3 – Donner la définition de la fonction `list_assocs` qui cette fois-ci donne la liste de toutes les valeurs associées à la clé donnée ou la liste vide si l'association n'existe pas.

Exemple
```
list_assocs 2 [(1, "one") ; (1, "un"); (2, "two") ; (2, "deux") ] = [ "two" ;
     "deux" ]
list_assocs 3 [(1, "one") ; (1, "un"); (2, "two") ; (2, "deux") ] = []
```

Q4 – Donner une définition de la fonction `list_combine : 'a list -> 'b list -> ('a * 'b) list` qui permet à partir de deux listes clés et valeurs de coupler les deux pour former une liste d'association. On supposera que les listes passées en arguments sont de même taille.

Exemple `list_combine [1;2;3] ["un";"deux";trois"] = [(1, "un"); (2, "deux"); (3, "trois")]`

Remarque : Vous pourrez vous servir ou non de la fonction `List.map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` qui, étant donnée une fonction `f` et deux listes `[x0; x1; ...; xk]` et `[y0; y1; ..;yk` de même taille, calcule la liste `[(f x0 y0); (f x1 y1); ...; (f xk yk)]`.

Exemple `List.map2 (fun a b -> a * b) [1;2;3;4] [4;3;2;1]` calcule la liste `[4;6;6;4]`

Q5 – On a, dans la bibliothèque standard, la fonction `List.exists : ('a -> bool) -> 'a list -> bool` telle que `(List.exists p xs)` donne `true` si la liste `xs` contient un élément `x` tel que `(p x)` vaut `true`; et donne `false` sinon.

Donnez la définition de la fonction `mem : 'a -> ('a * 'b) list -> bool` qui a pour valeur `true` si la clé est présente dans une table d'association. Utilisez `List.exists`.

Exercice V : Filtrage par motif en profondeur

On se donne une base de données de films sous la forme suivante :

```
let films =
[
("Pulp Fiction", ("Tarantino","Quentin"), 1994, [("Travolta","John")  ; ("Thurman","Uma")]);
("Psychose"    , ("Hitchcock","Alfred") , 1960, [("Perkins","Anthony"); ("Leigh","Janet")]);
("Shining"     , ("Kubrick","Stanley")  , 1980, [("Nicholson","Jack") ; ("Duvall","Shelley")]);
("Barry Lyndon", ("Kubrick","Stanley")  , 1975, [("Dullea","Keir")    ; ("Lockwood","Gary")]);
("Grease"      , ("Randal","Kleiser")   , 1978, [("Travolta","John")  ; ("Olivia","Newton-John")]);
]
```

Q1 – Quel est le type de `films` ? On le nommera `film_t`dans le reste de l'énoncé.

Q2 – En utilisant la fonction `List.map : ('a -> b') -> 'a list -> 'b list`, donnez la définition de la fonction `titres : film_t -> string list` qui donne la liste de tous les titres de films.

Q3 – En utilisant les fonctions `List.filter` et `List.map`, donnez la définition de la fonction `film_1980` de signature `film_t -> string list` qui calcule la liste des titres de tous les films sortis en 1980.

Q4 – En utilisant les fonctions `List.filter` et `List.map`, donnez la définition de la fonction `film_by_actor` de signature `string -> string -> film_t -> string list` qui, à partir du prénom et du nom d'un acteur, calcule la liste des titres de tous les films dans lesquels cet acteur a joué.

Exemple | `film_by_actor "John" "Travolta" films = ["Pulp Fiction"; "Grease"]`

Exercice VI : Schéma d'accumulation

On rappelle la définition du schéma d'accumulation à droite:

```
List.fold_right f [e1;e2;...en] e = f e1 (f e2 ...  (f en e))
```

Q1 – Donner la signature de la fonction `List.fold_right`.

Q2 – Donner une définition de la fonction `somme_list : int list -> int` calculant la somme des éléments d'une liste de nombres.

Q3 – Réimplanter les fonctions `list_map`, `list_filter` de la librairie standard à l'aide de `List.fold_right`. `list_filter` est une fonction prenant en argument un *prédicat* (une fonction de type `'a -> bool`) et une liste et calcule la liste des éléments sur lesquels le prédicat s'évalue à vrai.

Exemple

```
list_map (fun x -> x mod 2) [1;2;3;4;5;6] = [1;0;1;0;1;0]
list_filter (fun x -> x mod 2 = 0) [1;2;3;4;5;6] = [2;4;6]
```

Q4 – Donner une définition de `list_fold_right`.

On rappelle la définition du schéma d'accumulation à gauche:

```
List.fold_left f e [e1;e2;...en] = f (...  (f (f e e1) e2) ...)  en
```

Q5 – Donner la signature et une définition de `list_fold_left`. Que remarquez-vous?

Q6 – Réimplanter les fonctions `list_forall`, `list_exists` de la librairie standard à l'aide de `List.fold_left`. `list_forall` (resp. `list_exists`) est une fonction prenant en argument un prédicat et calcule `true` si chaque (resp. au moins un) élément de la liste satisfait le prédicat et `false` sinon.

Exemple

```
list_forall (fun x -> x mod 2 = 0) [1;2;3;4;5;6] = false
list_exists (fun x -> x mod 2 = 0) [1;2;3;4;5;6] = true
```

Q7 – Critiquer l'implantation proposée à la question précédente.

Travaux sur machines

Exercice VII : Tri fusion

Le tri par fusion repose sur l'utilisation d'une fonction d'interclassement de deux listes triées. La fonction d'interclassement construit son résultat en parcourant en parallèle les deux listes et en sélectionnant le plus petit des deux éléments en tête de liste. Le parcours se poursuit en enlevant l'élément sélectionné jusqu'à ce qu'une des deux listes soit épuisée.

Dans la suite on passera en argument aux fonctions qui implémentent le tri une fonction de comparaison `inf: 'a -> 'a -> bool` qui implémente une relation d'ordre totale ($\prec$) sur les éléments du type `'a` de la façon suivante $x \prec y \Leftrightarrow$ `inf` $x\ y$.

Q1 – Donner la définition récursive de la fonction `merge: ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list` qui, étant donné une fonction de comparaison et deux listes (supposées ordonnées pour la relation d'ordre représentée par la fonction de comparaison) fusionne les deux listes par interclassement pour obtenir une liste ordonnée contenant les éléments des deux listes passées en arguments.

Exemple | `merge (fun x y -> x < y) [0;2;4;8] [1;3;5;7]` donne `[0; 1; 2; 3; 4; 5; 7; 8]`

Q2 – Donner une définiton récursive terminale de `merge`.

Attention: pour des raisons d'efficacité, la liste fusionnée résultat sera accumulée en ordre inverse. On utilisera finalement `List.rev` pour inverser la liste à la fin du calcul.

Q3 – Donner la définition de la fonction `split: 'a list -> 'a list * 'a list` qui distribue les éléments de la liste passée en paramètre vers un couple de deux listes de taille identique (à +/- 1 élément près).

Exemple | `split [1;5;3;2;4] = ([1;3;4], [5;2])`

Q4 – Donner une définition récursive terminale de la fonction `split`

Q5 – En déduire la définition de la fonction `merge_sort: ('a -> 'a -> bool) -> 'a list -> 'a list` qui calcule une copie ordonnée (pour la relation d'ordre induite par la fonction de comparaison passée en argument à `merge_sort`) de son second argument.

Attention: Bien qu'il soit possible de définir `merge_sort` de manière récursive terminale il n'est pas demandé de le faire pour des raisons de difficulté.

La fonction de tri définie par `merge_sort` dépend d'une fonction de comparaison représentant une relation d'ordre. Dans la suite de cet exercice on se propose de définir la fonction de comparaison représentant l'ordre lexicographique sur les listes.

Q6 – Définir une fonction `padding: 'a list -> 'a list -> 'a -> 'a list * 'a list` qui, étant donnée deux listes l_1 et l_2 et un élément x calcule deux listes l'_1 et l'_2 obtenues par ajout de x à la fin de la liste la plus courte parmi l_1 et l_2 de sorte que l'_1 et l'_2 aient la même longueur.

Exemple | `padding [1;5;3] [2] 0 = ([1;5;3], [2;0;0])`

Q7 – Définir une fonction `lex: ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list -> bool` qui, étant donnée une fonction de comparaison `inf` sur le type `'a`, un élément minimal de type `'a` (minimal pour la relation d'ordre représentée par `inf`), calcule une fonction de comparaison représentant l'ordre lexicographique sur des listes d'éléments de type `'a`. La fonction `lex` devra utiliser la fonction `padding`.

Exemple

```
let inf_int_list = lex (fun x y -> x < y) min_int
inf_int_list [1; 5; 3] [0] = false
inf_int_list [1; 2; 5] [1; 3] = true
```

Q8 – En utilisant la fonction de comparaison des booléens suivante

```
let cmp_bool (b1:bool) (b2:bool) : bool =
   match b1, b2 with
     false, true -> true
   | _ -> false
```

et la fonction `lex`, définissez la fonction `sort_bool_list : ((bool list) list) -> ((bool list) list)` qui trie une liste de listes de booléens selon l'ordre lexicographique.