Examen partiel du 17 Novembre 2022

Durée 1h30

Téléphones, calculettes et ordinateurs interdits. Tous les documents sont autorisés. Dans chaque question, vous pouvez utiliser les fonctions demandées dans les questions et les exercices précédents même si vous ne les avez pas définies. Vous pouvez utiliser les fonctions prédéfinies de la librairie standard (par exemple `List.mem`*), sauf spécifié autrement dans la question. Le barème sur 27 est donné à titre indicatif : la note finale sur 20 sera* $\frac{n \times 20}{27}$ *en considérant le total* n *de points obtenus. Inscrire votre nom, votre numéro d'étudiant et le numéro de votre groupe de TD sur votre copie.*

Exercice 1 (1,5+1,5=3 points)**.**

1. Définir une fonction récursive de signature `somme_n (n:int) : int` qui calcule la somme $\sum_{i=0}^{n} i$ des `n` premiers entiers naturels. On suppose `n` positif ou nul.

```
# somme_n 0;;
- : int = 0
```

```
# somme_n 5;;
- : int = 15
```

Une solution

```
let rec somme_n (n:int) : int =
  if n <= 1 then n
  else n + (somme_n (n-1))
```

2. On souhaite numéroter chaque paire d'entiers naturels en associant à toute paire (x, y) l'entier $\left(\sum_{i=0}^{x+y} i\right) + x$. Définir une fonction de signature `pair ((x,y):int*int) : int` qui calcule le numéro de la paire d'entiers naturels `(x,y)`. On suppose `x` et `y` positifs ou nuls.

```
# pair (1,2);;
- : int = 7
```

```
# pair (2,1);;
- : int = 8
```

Une solution

```
let pair ((x,y):int*int) : int =
  (somme_n (x+y)) + x
```

Exercice 2 (1+2+2+2+3=10 points)**.**

1. Définir une fonction de signature `remove (x:'a) (l:'a list) : 'a list` qui construit la liste contenant les éléments de `l` qui sont différents de `x`. On supposera que la liste `l` ne contient pas de doublons (i.e. chaque élément de `l` apparaît une unique fois dans `l`).

```
# remove 2 [3;5;0;2;1];;
- : int list = [3;5;0;1]
```

```
# remove 7 [3;5;0;2;1];;
- : int list = [3;5;0;2;1]
```

```
# remove 4 [];;
- : int list = []
```

Une solution

```
let rec remove (x:'a) (l:'a list) : 'a list =
match l with
 | [] -> []
 | h :: t -> if h=x then t else h :: (remove x t)
```

2. Sans utiliser la fonction `List.filter`, définir une fonction de signature :

   `remove_pair (x:'a) (r:('a*'a) list) : ('a*'a) list`

   qui construit la liste contenant les paires `(x1,x2)` de la liste `r` tels que `x1` et `x2` sont différents de `x`. La liste `r` peut contenir plusieurs paires dans lesquelles `x` apparaît.

```
# remove_pair 2 [(1,3); (2,1); (4,9); (9,2); (2,2)];;
- : (int * int) list = [(1, 3); (4, 9)]
# remove_pair 2 [];;
- : (int * int) list = []
```

Une solution

```
let rec remove_pair (x:'a) (r:('a*'a) list) : ('a*'a) list =
  match r with
  | [] -> []
  | (h1,h2) :: t ->
    if h1=x || h2=x then remove_pair x t
                    else (h1,h2) :: (remove_pair x t)
```

3. Redéfinir la fonction `remove_pair` en utilisant la fonction `List.filter`.

Une solution

```
let remove_pair (x:'a) (r:('a*'a) list) : ('a*'a) list =
  List.filter (fun (x1,x2) -> (x1 <> x) && (x2 <> x)) r
```

4. En utilisant la fonction `List.map`, définir une fonction de signature :

   `map_cons (x:'a) (l: 'a list list) : 'a list list`

   qui construit la liste contenant les listes de la liste de listes `l` dans lesquelles l'élément `x` a été ajouté en tête de liste.

```
# map_cons 2 [];;
- : int list list = []
# map_cons 2 [[1; 3; 5]; [4; 1; 7; 9]];;
- : int list list = [[2; 1; 3; 5]; [2; 4; 1; 7; 9]]
```

Une solution

```
let map_cons (x:'a) (l: 'a list list) : 'a list list =
  List.map (fun y -> x :: y) l
```

5. Définir une fonction de signature `is_before (x:'a) (y:'a) (l:'a list) : bool` qui renvoie `true` si et seulement si l'élément `x` apparaît avant l'élément `y` dans la liste `l`. On pourra utiliser la fonction `List.mem`. On suppose ici que la liste `l` ne contient pas de doublons.

```
# is_before 'a' 'd' [];;
- : bool = false
# is_before 'a' 'd' ['b'; 'a'; 'k'; 'd'; 'p'];;
- : bool = true
# is_before 'a' 'd' ['b'; 's'; 'k'; 'd'; 'p'];;
```

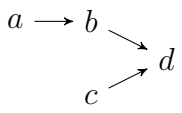
```
- : bool = false
# is_before 'a' 'd' ['b'; 'a'; 'k'; 's'; 'p'];;
- : bool = false
# is_before 'a' 'd' ['b'; 'd'; 'k'; 'a'; 'p'];;
- : bool = false
```

Une solution

```
let rec is_before (x:'a) (y:'a) (l:'a list) : bool =
  match l with
  | [] -> false
  | h :: t ->
    if h=x then List.mem y t
           else is_before x y t
```

Exercice 3 (2+2+2+2+2+4=14 points)**.**
Dans cet exercice, on pourra utiliser les fonctions définies dans l'exercice 2.

On considère une relation binaire, notée $\rightarrow$, définie sur un ensemble de tâches à effectuer : $t_1 \rightarrow t_2$ signifie que la tâche t_1 doit obligatoirement être effectuée avant la tâche t_2. D'autres tâches peuvent être effectuées entre les tâches t_1 et t_2. La relation $\rightarrow$ spécifie donc des contraintes sur l'ordre d'exécution d'un ensemble de tâches. Par exemple, la relation représentée ci-dessous exprime des contraintes sur les tâches a, b, c et d.



Cette relation spécifie que pour que la tâche b puisse être effectuée il faut que la tâche a ait déjà été effectuée et que pour que la tâche d puisse être effectuée il faut que les tâches b et c aient déjà été effectuées. Cette relation ne spécifie aucune contrainte entre les tâches b et c : on peut donc effectuer la tâche b avant la tâche c ou bien la tâche c avant la tâche b.

On définit la relation binaire $\rightarrow$ par une liste (sans doublons) contenant les paires (x, y) telles que $x \rightarrow y$. Par exemple, la relation représentée ci-dessus sera définie par la liste :

```
let ex_r = [('a','b'); ('b','d'); ('c','d')]
```

Le but de cet exercice est d'identifier et de construire les différentes façons d'exécuter un ensemble de tâches en respectant les contraintes de la relation $\rightarrow$. Il s'agit donc de considérer des listes (sans doublons) de tâches $\ell = [t_1; \cdots; t_n]$ telles que si $t_i \rightarrow t_j$ alors $j > i$: s'il existe une paire (t_i, t_j) dans la relation alors t_j apparaît après t_i dans la liste ℓ. Une liste qui vérifie cette propriété est appelée une extension de la relation $\rightarrow$. Par exemple, avec la relation `ex_r` définie ci-dessus, seules les trois listes `['a';'b';'c';'d']`, `['a';'c';'b';'d']` et `['c';'a';'b';'d']` sont des extensions de `ex_r` contenant les tâches `'a'`, `'b'`, `'c'`, et `'d'`.

1. Sans utiliser la fonction `List.for_all`, définir une fonction de signature :

   ```
   is_good (l:'a list) (r:('a*'a) list) : bool
   ```

   qui renvoie `true` si et seulement si la liste `l` est une extension de la relation `r`.

```
# is_good ['a';'c';'d'] ex_r;;
- : bool = false
# is_good ['c';'a';'b';'d'] ex_r;;
- : bool = true
# is_good ['a';'c';'b';'d'] ex_r;;
- : bool = true
# is_good [] ex_r;;
- : bool = false
```

```
# is_good ['a';'b';'c';'d'] ex_r;;
- : bool = true
# is_good ['a';'d';'b';'c'] ex_r;;
- : bool = false
# is_good ['a';'d';'b';'c'] [];;
- : bool = true
# is_good [] [];;
- : bool = true
```

Une solution

```
let rec is_good (l:'a list) (r:('a*'a) list) : bool =
  match r with
  | [] -> true
  | (h1,h2) :: t -> (is_before h1 h2 l) && (is_good l t)
```

2. Redéfinir la fonction `is_good` en utilisant la fonction `List.for_all`.

Une solution

```
let is_good (l:'a list) (r:('a*'a) list) : bool =
  List.for_all (fun (x,y) -> is_before x y l) r
```

3. Sans utiliser la fonction `List.exists`, définir une fonction de signature :

   `is_minimal (e:'a) (r:('a*'a) list) : bool`

   qui renvoie `true` si et seulement si la tâche `e` peut être effectuée avant toutes les autres tâches en respectant la relation `r`.

```
# is_minimal 'a' ex_r;;
- : bool = true
# is_minimal 'b' ex_r;;
- : bool = false
# is_minimal 'c' ex_r;;
- : bool = true
```

```
# is_minimal 'd' ex_r;;
- : bool = false
# is_minimal 'a' [];;
- : bool = true
# is_minimal 'x' ex_r;;
- : bool = true
```

Une solution

```
let rec is_minimal (e:'a) (r:('a*'a) list) : bool =
  match r with
  | [] -> true
  | (h1,h2) :: t -> (not (e=h2)) && (is_minimal e t)
```

4. Redéfinir la fonction `is_minimal` en utilisant la fonction `List.exists`.

Une solution

```
let is_minimal (e:'a) (r:('a*'a) list) : bool =
  not (List.exists (fun (x,y) -> y=e) r)
```

5. Définir une fonction de signature `minimaux (l:'a list) (r:('a*'a) list) : 'a list` qui construit la liste des tâches présentes dans `l` qui peuvent être effectuées avant toutes les autres tâches de `l` en respectant la relation `r`.

```
# minimaux ['a';'b';'c';'d'] ex_r;;
- : char list = ['a'; 'c']
# minimaux [] ex_r;;
- : char list = []
```

```
# minimaux ['a';'b';'c';'d'] [];;
- : char list = ['a'; 'b'; 'c'; 'd']
```

Une solution

```
let minimaux (l:'a list) (r:('a*'a) list) : 'a list  =
  List.filter (fun x -> is_minimal x r) l
```

6. Définir une fonction de signature `make_good (l:'a list) (r:('a*'a) list):'a list list` qui, étant donnée une liste de tâches `l` non vide, construit la liste de toutes les extensions de la relation `r` contenant les tâches de `l`. On suppose ici que toutes les tâches apparaissant dans la liste `r` sont présentes dans la liste `l`, mais que la liste `l` peut contenir des tâches n'apparaissant pas dans `r`.

```
# make_good ['a';'b';'c';'d'] ex_r;;
- : char list list =
[['a'; 'b'; 'c'; 'd']; ['a'; 'c'; 'b'; 'd']; ['c'; 'a'; 'b'; 'd']]
# make_good ['x';'y';'z'] [];;
- : char list list =
[['x'; 'y'; 'z']; ['x'; 'z'; 'y']; ['y'; 'x'; 'z']; ['y'; 'z'; 'x'];
 ['z'; 'x'; 'y']; ['z'; 'y'; 'x']]
```

Indications. Il pourra être judicieux de construire la liste des tâches qui peuvent être exécutées avant toutes les autres, et pour chaque tâche t de cette liste :

(1) de considérer (récursivement) les extensions de la relation `r` privée de toutes les paires contenant t pour l'ensemble des tâches de `l` privée de t

(2) d'ajouter la tâche t en tête de toutes les extensions obtenues en (1)

En plus des fonctions prédéfinies sur les listes vues en cours, on pourra utiliser avec profit la fonction prédéfinie `List.flatten` qui permet de concaténer les listes présentes dans une liste de listes :

```
# List.flatten [[1;2;3]; [7]; [2;5;9;4]];;
- : int list = [1; 2; 3; 7; 2; 5; 9; 4]
# List.flatten [[[1;2;6]; [2;3]]; [[7]]; [[2;5;9]; [4;3;4;5]; []; [4;5]]];;
- : int list list =
[[1; 2; 6]; [2; 3]; [7]; [2; 5; 9]; [4; 3; 4; 5]; []; [4; 5]]
```

Une solution

```
let rec make_good (l:'a list) (r:('a*'a) list) : 'a list list  =
  match l with
  | [e] -> [[e]]
  | l ->
    let lm = minimaux l r in
    List.flatten
      (List.map
         (fun m -> (map_cons m (make_good (remove m l) (remove_pair m r))))
         lm)
```