université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

mercredi 9 novembre 2022

Introduction à la programmation fonctionnelle
L2 Info./LDD2 IM

Examen

Durée 2h00, tiers temps additionnel 40 minutes

Consignes l'examen dure 2h00 et est sur 20 points. Aucun document n'est autorisé. Un aide mémoire OCaml est disponible à la page 4. Le barème est indicatif et proportionnel à la difficulté des exercices.

- il est interdit d'échanger du matériel entre étudiants
- déposez votre carte d'étudiant ou pièce d'identité sur la table, de façon à ce qu'elle puisse être lue par le surveillant
- les téléphones portables doivent être **éteints** et **rangés** dans les sacs. La présence d'un téléphone, même éteint sur votre table sera considérée comme une tentative de fraude.
- votre copie est ramassée à votre place par les surveillants.
- une fois la copie rendue, merci de quitter la salle sans provoquer d'attroupement et dans le respect de la distanciation

1 Lecture de code, typage (4 points)

Pour chacun des fragments de codes ci-dessous, donner :

- le type des variables globales (introduites par **let** sans « **in** »), qu'il s'agisse de fonction ou de simples valeurs. Par exemple pour la question 1, on demande le type de la fonction `f` et celui de la variable `s`.
- les erreurs de types ainsi qu'une explication succincte de l'erreur (une phrase max). Vous pouvez ignorer tout code se trouvant après une erreur de type (comme le compilateur OCaml, vous vous arrêtez à la première erreur de typage).

Questions

(1)

```
let f x y = (x + y, x - y)
let s = snd (f 4 5)
```

(2)

```
let g y = y *. 2.0
let gl =
  List.map g [ "45.0" ]
```

(3)

```
let rec h l =
  match l with
    [] -> []
  | (p, e) :: ll ->
    if p then e :: (h ll)
    else h ll

let m lst =
 let lst2 =
  List.map
   (fun x -> (x mod 2 = 0, x)) lst
 in h lst2
```

2 Fonctions récursives et filtrage (7 points)

Dans cet exercice, on écrira des fonctions récursives, sans utiliser les itérateurs du module `List`. Il est évidemment autorisé (et parfois nécessaire) d'utiliser des fonctions auxiliaires.

Questions

1. (2 points) Écrire une fonction `choose: ('a -> bool) -> ('a * 'a) list -> 'a list` telle que `choose f l` renvoie la liste des composantes des paires de `l` pour lesquelles `f` renvoie vrai. Par exemple :

   ```
   choose (fun x -> x mod 2 = 0) [ (0, 1); (2, 4); (5, 7); (8, 9)]
   ```

   renvoie la liste

   ```
   [ 0; 2; 4; 8 ]
   ```

   L'ordre dans la liste résultat doit être le même que dans la liste initiale (on considère que le membre droit d'une paire arrive après le membre gauche).

2. (2 points) Écrire une fonction `popcount : int -> int` qui compte le nombre de bits valant 1 dans la représentation binaire de son argument. Vous **devez** écrire une fonction auxiliaire qui doit être récursive terminale. Voici des exemples de résultats d'appels à `popcount` :
   — `popcount 0` donne `0`
   — `popcount 8` donne `1` (car 8 s'écrit `1000` en base 2).
   — `popcount 5` donne `2` (car 5 s'écrit `101` en base 2).
   — `popcount 15` donne `4` (car 15 s'écrit `1111` en base 2).
   — `popcount 37` donne `3` car 37 s'écrit `100101` en base 2).
   Rappel : on peut tester le bit de poids faible d'un entier en testant si cet entier est divisible par 2 (avec `mod`) et on peut supprimer ce bit de poids faible (i.e. décaler tous les bits d'un cran vers la droite) en divisant le nombre par 2.

3. (3 points) On se donne le type suivant représentant des cartes de tarot :

   ```
   type carte = Excuse | Atout of int
              | Roi | Dame | Cavalier | Valet
              | Valeur of int
   ```

   (a) (1.5 point) Écrire une fonction `score : carte -> float` qui calcule le nombre de points d'une carte :
   — l'`Excuse`, l'`Atout(21)`, l'`Atout(1)` et le `Roi` rapportent `4.5` points.
   — la `Dame` rapporte `3.5` points
   — le `Cavalier` rapporte `2.5` points
   — le `Valet` rapporte `1.5` points
   — les autres cartes (i.e. les autres atouts et les cartes de valeur) rapportent `0.5` point.

   (b) (1.5 point) Écrire une fonction `score_total : carte list -> float` qui calcule le score d'une liste de cartes. Vous pouvez réutiliser la fonction `score` ci-dessus même si vous n'avez pas réussi à l'écrire.

3 Musique (9 points)

On considère une application permettant d'écouter de la musique. Un morceau (*track* en anglais) est représenté par un titre, une durée en secondes, un album et un artiste. La *file de lecture* (*play queue* en anglais) est une structure contenant la liste des morceaux à venir et la liste des morceaux déjà joués. On se donne les types OCaml définis ci-dessous :

```
type track = { title : string;
               length : int;  (* en secondes *)
               album : string;
               artist : string }

type play_queue = { prev_tracks : track list;
                    next_tracks : track list }
```

Le type `play_queue` représente la file de lecture. Le champ `next_tracks` donne la liste des morceaux à venir (et par convention, le premier morceau de cette liste est le morceau en cours de lecture). Le champ `prev_tracks` représente la liste des morceaux **déjà lus**, donnés du plus récemment lu au plus ancien. Ainsi, lorsque le premier morceau de `next_tracks` est terminé, il est retiré et placé **en tête** de la liste `prev_tracks`.

Dans toute la suite :

— il est fortement suggéré d'utiliser les fonctions de la bibliothèque standard sur les listes afin de gagner du temps, mais ce n'est pas une obligation ;

— si vous n'arrivez pas à écrire une fonction, vous pouvez quand même l'utiliser dans les questions suivantes afin de ne pas être pénalisé plusieurs fois.

Une fonction correcte donnera tous les points, indépendamment de sa complexité. Enfin, la fonction `List.rev` permettant de renverser une liste a été **ajoutée à l'aide mémoire**.

Questions

1. (1 point) écrire une fonction `pr_track : track -> unit` qui affiche dans la console un morceau au format suivant `"album: titre (artiste) [duree]\n"`.
2. (0.5 point) écrire une fonction `pr_track_list : track list -> unit` qui affiche dans la console la liste des morceaux donnée en argument.
3. (1 point) écrire une fonction `pr_play_queue : play_queue -> unit` qui affiche dans la console la chaine

   `"Previous tracks:\n"`

   suivie de la liste des morceaux déjà joués, du plus récent au plus ancien puis la chaine

   `"Next tracks:\n"`

   suivie de la liste des morceaux restants, dans l'ordre où ils vont être joués.
4. (1 point) écrire une fonction `next : play_queue -> play_queue` qui renvoie une nouvelle file de lecture dans laquelle on est passé au morceau suivant. Si on était sur le dernier morceau (*i.e.* si le champ `next_tracks` ne contient qu'un élément), la fonction renvoie son argument.
5. (1 point) écrire une fonction `next_loop : play_queue -> play_queue`. S'il reste des morceaux à jouer, la fonction se comporte comme `next`. Sinon, la liste est ré-initialisée pour contenir tous les morceaux déjà joués, remis dans l'ordre (i.e. du plus ancien au plus récemment joué).
6. (1 point) écrire une fonction `remaining_time : play_queue -> int` qui renvoie le nombre de secondes restant à jouer (on compte la durée complète du morceau en cours).
7. (1.5 point) écrire une fonction `shuffle : play_queue -> play_queue` qui renvoie la file de lecture contenant tous les morceaux (à jouer et déjà joués) de la file passée en argument mélangés aléatoirement.

   Indication : on peut mélanger une liste aléatoirement en

   — associant à chaque élément un nombre flottant aléatoire compris entre 0 et 1 ;

   — en triant la liste selon ce nombre ;

   — puis en retirant le nombre de la liste triée.

   Un nombre aléatoire compris entre 0 et 1 peut être obtenu par `Random.float 1.0`.
8. (2 points) écrire une fonction `tidy : play_queue -> (string * track list) list` prenant en argument une file de lecture et qui renvoie une liste de paires. Chaque paire est composé d'un nom d'album et de la liste des morceaux de cet album présent dans la file de lecture (soit dans les morceaux joués, soit dans les morceaux à venir).

   Indication : une façon de faire est d'abord de trier la liste de tous les morceaux par album. Ainsi tous les morceaux d'un même album se suivent dans la liste ainsi triée. Cette liste peut ensuite être parcourue dans l'ordre pour former les « paquets » de morceaux d'un même album.

Aide-mémoire OCaml

Cet aide mémoire rappelle les types de bases en OCaml ainsi que les fonctions utilitaires associées ainsi que leurs types. Attention, toutes ces fonctions ne sont pas forcément utiles pour les exercices. Dans la suite, lorsqu'une fonction est marquée comme « opérateur binaire » (par exemple `+`) cela signifie qu'il faut l'écrire `a op b`. Sinon c'est une fonction qu'il faut appeler avec `op a b`.

Entiers

Le type `int` représente des entiers signés. Les constantes entières s'écrivent simplement `0`, `42`, `-233`. Les opérations et fonctions sur les entiers sont :

`+ : int -> int -> int` addition entre deux entiers (opérateur binaire).
`- : int -> int -> int` soustraction entre deux entiers (opérateur binaire).
`* : int -> int -> int` multiplication entre deux entiers (opérateur binaire).
`/ : int -> int -> int` division **entière** entre deux entiers (opérateur binaire).
`mod : int -> int -> int` reste dans la division entière (opérateur binaire).
`int_of_string : string -> int` conversion d'une chaîne en entier. Lève une exception si la chaîne n'est pas au bon format.
`int_of_float : float -> int` conversion d'un flottant en entier (la partie décimale est tronquée).

Flottants

Le type `float` représente des nombre flottants. Les constantes flottantes s'écrivent en notation scientifique `0.5`, `42e3`, `-233.8e-20`. Les opérations et fonctions sur les flottants sont :

`+. : float -> float -> float` addition entre deux flottants (opérateur binaire).
`-. : float -> float -> float` soustraction entre deux flottants (opérateur binaire)
`*. : float -> float -> float` multiplication entre deux flottants (opérateur binaire)
`/. : float -> float -> float` division entre deux flottants (opérateur binaire)
`** : float -> float -> float` puissance entre deux flottants (opérateur binaire)
`float_of_string : string -> float` conversion d'une chaîne en flottant. Lève une exception si la chaîne n'est pas au bon format.
`float : int -> float` conversion d'un flottant en entier (la partie décimale est tronquée).
`sqrt : float -> float` racine carrée d'un flottant.

Booléens

Le type `bool` représente des booléens. Les constantes booléennes sont `true` et `false`. Les opérations et fonctions sur les booléens sont :

`&& : bool -> bool -> bool` « et » logique entre deux booléens (opérateur binaire).
`|| : bool -> bool -> bool` « ou » logique entre deux booléens (opérateur binaire).
`not : bool -> bool` négation d'un booléen.

Chaînes de caractères

Le type `string` représente des chaînes de caractères. Les chaînes de caractères constantes sont délimitées par des guillemets :`"Hello, world !"`. De façon usuelle, la séquence d'échappement `\n` représente un retour à la ligne. Les opérations et fonctions sur les chaînes sont :

`^ : string -> string -> string` concaténation entre deux chaînes (opérateur binaire).
`String.length : string -> int` longueur d'une chaîne.

`String.trim : string -> string` renvoie une copie de la chaîne où les blancs (espaces, tabulations, retours à la ligne) en début et en fin de chaîne ont été supprimés.

Affichage

La fonction `Printf.printf fmt arg1 arg2 ... argn`, permet d'afficher `n` arguments en utilisant la chaîne de format `fmt`. Cette dernière est une chaîne de caractères contenant des séquences spéciales :
`%d` Affichage d'un entier.
`%s` Affichage d'une chaîne.
`%f` Affichage d'un flottant.

Exemple : `Printf.printf "Salut, mon nom est %s et j'ai %d ans""Toto"42` affiche :

```
Salut, mon nom est Toto et j'ai 42 ans
```

Comparaisons

En OCaml les opérations de comparaison permettent de comparer n'importe quelles valeurs du même type :
`<, <=, >, >=, =, <> : 'a -> 'a -> bool` comparaisons entre deux valeurs (respectivement, inférieur, inférieur ou égal, supérieur, supérieur ou égal, égal et différent), (opérateur binaire).
`compare : 'a -> 'a -> int` comparaison générique : `compare x y` renvoie un entier négatif si `x < y`, nul si `x = y` et positif si `x > y`.
`min : 'a -> 'a -> 'a` renvoie la plus petite de deux valeurs du même type.
`max : 'a -> 'a -> 'a` renvoie la plus grande de deux valeurs du même type.

n-uplets

Les n-uplets ou produits sont délimités par des parenthèses et des virgules. Dans le cas particulier des paires, deux fonctions `fst` et `snd` permettent d'accéder à la première et seconde composante. Dans les autres cas, on peut utiliser un `let` multiple :

```
let p1 = (10, 12);;
let p2 = (-1, false);;
let t3 = ("A", "B", 24);;

let x = fst p1;; (* 10 *)
let y = snd p2;; (* false *)
let a, b, n = t3;;  (* a vaut "A", b vaut "B" et n vaut 24 *)
```

Définitions de types

La directive `type t = ...` permet de définir un type OCaml nommé `t`. Ce type peut être :

Un produit nommé est défini en donnant pour chaque étiquette le type des valeurs associées :

```
type point_colore = { x : float; y : float; couleur : string }
```

On peut créer des valeurs enregistrement avec des accolades et accéder aux champs avec la notation `.f` :

```
let prouge = { x = 0.5; y = 10.3; couleur = "rouge" } ;;

(* Fonction pour afficher un point coloré *)
let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
        p.x p.y p.couleur;;
```

Un type somme est défini en donnant la liste de cas possibles :

```
type val_carte = Roi | Dame | Valet | Val of int
```

On peut créer des valeurs de ces types en utilisant les constantes ou en leur donnant un argument. On peut tester les valeurs en utilisant l'opérateur de filtrage :

```
let dix = Val 10;;
let as = Val 1;;

(* Fonction pour afficher une valeur de carte *)
let pr_val v = match v with
      Roi -> Printf.printf "%s"  "Roi"
    | Dame -> Printf.printf "%s"  "Dame"
    | Valet -> Printf.printf "%s"  "Valet"
    | Val (1) -> Printf.printf "%s"  "As"
    | Val (n) -> Printf.printf "%d"  n;;
```

Exceptions

En OCaml, les exceptions sont des objets particuliers permettant de signaler une erreur. On peut définir une exception avec la directive **exception** `E` **of** `...` :

```
exception MonErreur of string (* un message *)
```

On peut « lever » une exception, c'est à dire signaler une erreur au moyen de la fonction prédéfinie `raise` :

```
let err_arg_invalide () = raise (MonErreur "argument invalide");;
```

Une exception non rattrapée interrompt immédiatement le programme. On peut rattraper une exception avec la construction **try with** :

```
try
  18 + (f 42)  (* f peut lever une exception *)
with
   Not_found -> (* si l'exception Not_found est levée par f *)
                10
 | MonErreur msg ->
                12
```

Si on veut signaler une erreur avec un message, la fonction prédéfinie `failwith` permet de lever une exception avec ce message en argument.

```
if x < 0.0 then
     failwith "valeur négative interdite"
else
    sqrt x;;
```

Listes

Le type OCaml des listes est `'a list` et permet de représenter une collection ordonnée de valeurs du type `'a`. Les listes constantes sont délimitées par des crochets et des points-virgules : `[1; 2; 42; -5]`. La liste vide est représentée par `[]`. Les opérations et fonctions sur les listes sont :

`:: : 'a -> 'a list -> 'a list` ajout en tête de liste : `1 :: l` (opérateur binaire).

`@ : 'a list -> 'a list -> 'a list` concaténation de deux listes.

`List.rev : 'a list -> 'a list` renvoie la liste avec les éléments dans l'ordre inverse.

`List.iter : ('a -> unit) -> 'a list -> unit` `List.iter f l` applique `f` à tous les éléments de `l`. La fonction `f` ne renvoie pas de résultat (par exemple elle fait un affichage).

`List.map : ('a -> 'b) -> 'a list -> 'b list` `List.map f l` applique `f` à tous les éléments de `l` et renvoie la liste des images par `f`.

`List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` `List.fold_left f init l` applique la fonction de combinaison `f` à `init` et tous les éléments de `l` dans l'ordre. Si

```
l = [ v1; v2; ... ; vn]
```

alors

```
List.fold_left f init l = (f ... (f (init v1) v2) ... vn)
```

`List.filter : ('a -> bool) -> 'a list -> 'a list` `List.filter f l` renvoie la liste de tous les éléments de `l` pour lesquels `f` renvoie **`true`**.

`List.assoc : 'a -> ('a * 'b) list -> 'b` `List.assoc a l` renvoie la seconde composante de la première paire dans `l` qui possède `a` comme première composante. La fonction lève l'exception `Not_found` si une telle paire n'existe pas.

`List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` `List.sort f l` renvoie une copie triée de `l` selon la fonction de comparaison `f`. Cette dernière suit les mêmes conventions que la fonction prédéfinie `compare`.

Enfin, on peut inspecter les listes au moyen de l'opérateur de filtrage :

```
1      (* teste si une liste est de longueur paire *)
2    let rec liste_long_paire l =
3      match l with
4      []    -> true (* liste vide est de longueur 0, pair *)
5    | [ _ ] -> false (* liste à un élément est de longueur 1, imapair *)
6    | _ :: _ :: ll -> liste_long_paire ll (* cas récursif *)
7  ;;
```