université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

mercredi 5 novembre 2025

Intro. à la prog. fonctionnelle
L2 Info./LDD2 IM

Examen

Durée 2h00, tiers temps additionnel 40 minutes

Consignes l'examen dure 2h00 et est sur 20 points. Les notes personnelles sont autorisés. Le sujet se termine sur la page 3. Un aide mémoire OCaml est disponible à la page 5 (vous pouvez le détacher du sujet pour le consulter pendant votre lecture). Le barème est indicatif et proportionnel à la difficulté des exercices.

- votre copie doit être anonymée
- 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, conformément au règlement des études.

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`, pas le type de la variable locale `sy`.
- 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 =
 let sy = string_of_int y in
  x ^ ", " ^ sy

let s = f 10 10
```

(2)

```
let g y = (y,y)

let gl =
  List.map g [1; 3; 4]
```

(3)

```
let rec h p l =
  match l with
    [] -> []
  | x :: ll ->
    if p x 5 then x :: (h p ll)
    else h p ll

let r =
  h (fun x y -> String.length x > y)
    [ "Hello"; "everybody" ]
```

2 Fonctions récursives et filtrage (6 points)

Dans cet exercice, on écrira des fonctions récursives, **sans utiliser** le module `List` ni l'opérateur `@`. Il est évidemment autorisé (et parfois nécessaire) d'utiliser des fonctions auxiliaires.

Questions

1. (2.5 points) Écrire une fonction `filter_int : (int -> bool) -> int -> int -> int list` telle que `filter_int f i j` renvoie la liste des entiers compris entre `i` et `j` inclus pour lesquels `f` renvoie `vrai`. Le résultat doit être trié par ordre croissant et la fonction doit être récursive terminale (ou appeler une fonction auxiliaire récursive terminale). Si `j < i`, la fonction renvoie la liste vide.

   ```
   filter_int (fun x -> x mod 3 = 0) 0 10
   ```

   renvoie la liste

   ```
   [ 0; 3; 6; 9 ]
   ```

2. (2 points) On considère le type de données :

   ```
   type op = Del | Dup | Nop
   ```

   Écrire une fonction `apply_op : 'a list -> op list -> 'a list` qui prend en argument une liste d'éléments et une liste de valeurs de type `op` supposées être de même longueur. Étant données deux listes `lv = [ v1; ...; vn ]` et `lo = [ o1; ...; on ]`, `apply_op lv lo` renvoie une liste où :
   — l'élément `vi` est dupliqué si `oi` vaut `Dup`
   — l'élément `vi` est supprimé si `oi` vaut `Del`
   — l'élément `vi` est laissé tel quel si `oi` vaut `Nop`
   Par exemple :

   ```
   apply_op [ 1; 2; 3; 4; 5 ] [ Dup; Del; Dup; Nop; Del ]
   ```

   renvoie

   ```
   [ 1; 1; 3; 3; 4 ]
   ```

   Les éléments de la liste résultante doivent être dans le même ordre que ceux de la liste initiale. Si les deux listes n'ont pas la même longueur, votre fonction doit lever une erreur avec `failwith "erreur"`. **On ne demande pas** que cette fonction soit récursive terminale.

3. (0.5 point) La fonction `f` ci-dessous est-elle récursive terminale (justifier brièvement) ?

   ```
   let rec f g l =
   match l with
     | [] -> []
     | e :: ll -> g e (f g ll)
   ```

4. (1 point) Donner le type de la fonction `f` ci-dessus.

3 Contacts (10 points)

On considère une application de carnet d'adresses (comme par exemple celle disponible sur un téléphone). Un tel carnet d'adresses est constitué d'une liste de contacts. On se donne les types OCaml définis à la figure 1. Le type `coord` représente des coordonnées d'un contact. Une coordonnée peut être soit un numéro de téléphone (représenté par une chaîne de caractères), soit un email (représenté par une chaîne de caractères), soit un identifiant de réseau social composé de deux chaînes de caractères (le nom du site, et l'identifiant sur le site, par exemple : `"Y (ex-Gwitter.com)"`, `"@chipmunk"`).

Un `contact` est un type enregistrement donnant le prénom et le nom d'un contact, la liste des coordonnées qui lui sont associées et un booléen indiquant si ce contact est un favori ou pas.

Enfin, un carnet d'adresse est simplement une liste de contacts. 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.

```
type coord =
 Tel of string
| Email of string
| Social of string * string

type contact = {
  prenom : string;
  nom : string;
  coords : coord list;
  favori : bool }

type carnet = contact list
```

FIGURE 1 – Les types représentant un carnet d'adresses

Questions

1. (1 point) Écrire une fonction `pr_coord : coord -> unit` qui affiche dans la console. La coordonnée est affiché de la façon suivante :
   — 4 caractères espaces
   — la chaîne de caractères : `"tel : "`, `"email : "` ou `"social : "`
   — la chaîne de caractères contenue dans la valeur pour les cas `Tel` et `Email`), les deux chaînes séparées par un espace pour le cas `Social`
   — un retour à la ligne
2. (1 point) Écrire une fonction `pr_contact : contact -> unit` qui affiche un contact dans la console, en affichant le prénom, suivit du nom séparé par un espace sur une ligne, la chaîne de caractère `"Favori"` si le contact est un favori sur une seule ligne et enfin la liste des coordonnées, un par ligne (en utilisant la fonction `pr_coord`).
3. (0.5 point) écrire une fonction `pr_carnet : carnet -> unit` qui affiche la liste des contacts dans la console (en utilisant la fonction `pr_contact`).
4. (1.5 point) Écrire une fonction `contact_max_coords : carnet -> contact` qui renvoie le contact ayant le plus de coordonnées dans le carnet d'adresses. S'il y a plusieurs contacts dans ce cas, votre fonction peut renvoyer n'importe lequel parmi ceux-ci. Si le carnet d'adresse est vide, la fonction lève une exception avec `failwith "erreur"`. La fonction `List.length : 'a list -> int` renvoie la longueur d'une liste.
5. (2 points) Écrire une fonction `fusion_coords : coord list -> coord list -> coord list`. Cette dernière fusionne deux listes de coordonnées en retirant les doublons éventuels.

   Indication : comme vu en cours, on pourra dans un premier temps trier les deux listes données en argument, puis les parcourir en même temps pour créer la liste fusionnée. La fonction prédéfinie `compare : 'a -> 'a -> int` peut être utilisée pour comparer deux coordonnées (un `Tel` sera inférieur à un `Email` qui lui même sera inférieur à un `Social` et deux coordonnées de même sorte seront comparées selon les chaînes de caractères de leur contenu). La fonction `compare` est décrite dans l'aide-mémoire, dans la section sur les comparaisons.
6. (2 points) Écrire une fonction `fusion_contacts : carnet -> carnet` qui fusionne les contacts de même nom et prénoms. La fusion de deux contacts donne un nouveau contact qui contient la fusion des coordonnées et est un favori si l'un des deux contacts était un favori.
7. (2 points) Écrire une fonction `effacer_social : string -> carnet -> carnet` qui prend un nom de réseau social et un carnet d'adresses et qui renvoie une copie du carnet dans lequel tous les contacts qui ne sont pas des favoris et dont l'une des coordonnées est sur le réseau social dont le nom est donné en argument sont supprimés.

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  ;;
```