université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

mercredi 6 novembre 2024

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. Aucun document n'est autorisé. Le sujet se termine sur la page 4. 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
— 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`.
— 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 z = (x /. z, y *. z)
let u = f true true false
```

(2)

```
let g a b = a * b
let gl =
  List.map (g 2) [ 3; 4; 5 ]
```

(3)

```
let rec h m l =
  match l with
    [] -> []
  | (x, y) :: ll ->
    (m x y)::(h m ll)

let r =
 h (fun x y -> (float x) +. y)
   [ (10, 2.5); (11, -1.4) ]
```

2 Fonctions récursives et filtrage (6 points)

Dans cet exercice, il est **interdit** d'utiliser les fonctions du module `List` ou l'opérateur de concaténation de liste `@`. Il est évidemment autorisé (et parfois nécessaire) d'utiliser des fonctions auxiliaires.

Questions

1. (2.5 points) Écrire une fonction `count_ok : (int -> bool) -> int -> int -> int` telle que `count_ok f i j` renvoie le nombre d'entiers compris entre `i` et `j` inclus pour lesquels `f` renvoie `true`. La fonction doit être récursive terminale (ou appeler une fonction auxiliaire récursive terminale). Si `j < i`, la fonction renvoie 0.

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

   renvoie `4`, car il y a 4 entiers (`0`, `3`, `6` et `9`) qui sont multiples de trois dans l'intervalle.

2. (2.5 points) On considère le type des opérations :

   ```
   type op = Add | Sub | Mul | Div
   ```

   Écrire une fonction `apply_op : (int * int) list -> op list -> int list` qui prend en argument une liste de paires d'entiers. La fonction renvoie la liste des résultats d'opérations entre les deux composantes de chaque paires, en respectant l'ordre initial. Par exemple :

   `apply_op [ (1, 2); (3, 3); (4, 5); (3, 2); (0, 4) ] [ Add; Mul; Sub; Div; Sub ]`

   renvoie

   ```
   [ 3; 9; -1; 1; -4 ]
   ```

   Si les deux listes n'ont pas la même longueur, votre fonction doit lever une exception avec l'instruction `failwith "erreur"`. **On ne demande pas** que cette fonction soit récursive terminale. Attention pour la division (cas `(3,2)` de l'exemple, on effectue la division entière avec « `/` »).

3. (1 point) Soit la fonction `split : ('a -> bool) -> 'a list -> 'a list * 'a list` :

   ```
   let split f l =
     let rec split_aux l acc1 acc2 =
       match l with
         [] -> (acc1, acc2)
       | e :: ll ->
             if f e then split_aux ll (e::acc1) acc2
                    else split_aux ll acc1 (e::acc2)
     in
      split_aux l [] []
   ```

   La fonction interne `split_aux` est-elle récursive terminale ? (justifier brièvement).

3 Inventaire (10 points)

On considère une application de gestion des produits vendus par un site de commerce en ligne. On se donne les types OCaml définis à la figure 1. Le type `produit` dénote un produit en donnant

```
type produit = { nom : string; marque : string; prix : float }
type stock = (produit * int) list
```

FIGURE 1 – Les types représentant les notes d'étudiants

son `nom`, sa `marque` et son `prix` (un flottant qu'on supposera positif).

Le stock global du magasin est une liste de paires `produit`, `quantite`.

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.

Questions

1. (1 point) Écrire une fonction `pr_produit : coord -> unit` qui affiche dans la console un produit de la façon suivante :
   — son nom, suivi d'un espace
   — sa marque entre parenthèses suivie d'un espace
   — le prix, suivi de `euro` si le prix est strictement inférieur à `2.0` et `euros` sinon
2. (1 point) Écrire une fonction `pr_stock : produit -> unit` l'inventaire dans la console, en affichant chaque produit (avec la fonction `pr_produit`) suivi d'un espace, suivi de sa quantité, suivi d'un retour à la ligne.
3. (1.5 point) Écrire une fonction `plus_cher : stock -> produit * int` qui prend en argument un stock et renvoie le produit dont le prix cumulé est le plus cher. Par exemple, pour le stock :

   ```
   [
     ({ nom = "basket"; marque = "Mike"; prix = 120.0 }, 20);
     ({ nom = "Jphone 17"; marque = "Affle"; prix = 1000.0 }, 2);
     ({ nom = "briquet"; marque = "Zhippo"; prix = 10.0 }, 400)
   ]
   ```

   La fonction renvoie `({ nom = "Briquet"; marque = "Zhippo"; prix = 10.0 }, 400)` car 4000 est supérieur à 2400 (prix cumulé du premier produit) et 2000 (prix cumulé du deuxième produit). **Vous pouvez supposer que tous les produits sont distincts**.
4. (1.5 points) Écrire une fonction `comp_prod : produit*int -> produit*int -> int` qui compare deux couples (produit, quantité) en comparant dans l'ordre :
   — par marque, puis en cas d'égalité
   — par nom, puis en cas d'égalité
   — par prix, puis en cas d'égalité
   — par quantité

   Votre fonction doit renvoyer un entier négatif, nul ou positif, selon que le premier argument est inférieur, égal ou supérieur au second.

   Indication : La fonction prédéfinie `compare : 'a -> 'a -> int` peut être utilisée pour comparer des chaînes de caractères, des flottants ou des entiers. La fonction `compare` est décrite dans l'aide-mémoire, dans la section sur les comparaisons.
5. (2.5 points) Écrire une fonction `compact_stock : stock -> stock` qui prend en argument un stock dans lequel peuvent se trouver plusieurs fois des produits de même nom et marque, mais avec des prix et quantité différentes. Votre fonction doit renvoyer un stock « compacté » dans lequel les produits de même noms et marques seront fusionné, en faisant la somme de leurs quantités et en prenant le maximum de leurs prix. Par exemple, pour le stock

   ```
   [
     ({ nom = "basket"; marque = "Mike"; prix = 120.0 }, 20);
     ({ nom = "Jphone 17"; marque = "Affle"; prix = 1000.0 }, 2);
     ({ nom = "basket"; marque = "Mike"; prix = 125.0 }, 10);
     ({ nom = "briquet"; marque = "Zhippo"; prix = 10.0 }, 400);
     ({ nom = "Jphone 17"; marque = "Affle"; prix = 990.0 }, 2);
   ]
   ```

   Le résultat de `compact_stock` est

```
[
  ({ nom = "basket"; marque = "Mike"; prix = 125.0 }, 30);
  ({ nom = "briquet"; marque = "Zhippo"; prix = 10.0 }, 400);
  ({ nom = "Jphone 17"; marque = "Affle"; prix = 1000.0 }, 4);
]
```

Indication : comme vu en cours, on pourra d'abord trier la liste en utilisant `comp_prod` puis parcourir la liste triée en fusionnant les produits identiques (qui se trouveront côte à côte dans la liste triée).

6. (2.5 points) Écrire une fonction `prod_par_marque : stock -> (string * produit) list` qui prend en argument un stock et renvoie une liste de paires. Chaque paire est le nom d'une marque et le produit le plus cher de cette marque.

   Indication on pourra écrire dans un premier temps une fonction auxiliaire

   ```
   ajout_prod : produit -> (string * produit) list -> (string * produit) list
   ```

   prenant en argument un produit `p` et une liste `l` de paires nom de marque et produit et qui fonctionne comme suit :
   — si `p.marque` n'apparait pas dans la liste `l`, alors `(p.marque, p)` est ajouté
   — si `(p.marque, q)` est dans la liste pour un certain produit `q` alors si `p.prix` est supérieur, on remplace `q` par `p` sinon on laisse `q`.

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 *)
```