université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

vendredi 7 janvier 2022

Introduction à la programmation fonctionnelle
L2 Informatique

Examen

Durée 2h00, tiers temps additionnel 40 minutes

Consignes l'examen dure 2h00 et est sur 20 points. Les notes manuscrites sont autorisées (1 feuille par support de cours, sans photocopie ni impression). Un aide mémoire OCaml est disponible à la page 5. Le barème est indicatif et proportionnel à la difficulté des exercices.

Conditions spéciales d'examen en raison de la situation sanitaire et des règles en vigueur :

- le port du masque en salle d'examen comme sur le reste du campus est obligatoire
- ne pas anonymiser vos copies
- il est interdit d'échanger du matériel entre étudiants
- signer une fois la feuille d'émargement avec votre stylo personnel, lors du passage du surveillant
- déposez votre carte d'étudiant ou pièce d'identité sur la table, de façon à ce qu'elle puisse être lue par le surveillant
- 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

Le non respect de ces règles peut entrainer l'exclusion **immédiate** de la salle d'examen.

1 Lecture de code, typage (4 points)

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

- soit le type de toutes les variables globales
- soit uniquement une erreur de type à l'endroit où elle se produit.

On ne demande pas de justification dans le premier cas. Les deux exemples ci-dessous illustrent le type de réponse attendue :

Exemple 1

```
1 let f y = y + 1;;
2 let x = f (f 42);;
```

Exemple 2

```
1 let f x = x + 1;;
2 let g y = "42" - (f y) ;;
```

Pour l'exemple 1 on attend la réponse :

- `f : int -> int`
- `x : int`

Pour l'exemple 2 on attend la réponse :

- erreur de typage ligne 2 : la soustraction attend 2 entiers mais est appliquée à une chaîne et un entier.

Questions

(1)

```
1 let f y = (string_of_int y) ^ "0";;
2 let y = int_of_string (f  42);;
```

(2)

```
1 let f y = 1 + y;;
2 let g y = (f y) > 0 ;;
3 let r = List.filter g [ 4; "5" ];;
```

(3)

```
let rec zip_map f l1 l2 =
 match l1, l2 with
  [], [] -> []
 |  ([], _) | (_, []) -> failwith "erreur"
 | x1 :: ll1, x2 :: ll2 ->
       (f x1, f x2) :: (zip_map f ll1 ll2)
     ;;

let l = zip_map float [1; 2; 3] [4; 5; 6] ;;
```

2 Fonctions récursives (6 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 (indépendantes ou imbriquées).

1. (2 point) écrire une fonction récursive `sum_if : (int -> bool) -> int -> int` telle que `sum_if f n` renvoie la somme de tous les entiers compris entre `0` et `n` pour lesquels `f` renvoie vrai.
2. (1 point) écrire une fonction `dupl : 'a list -> 'a list` qui « duplique » tous les éléments de la liste passée en argument. Par exemple `dupl [1;2;3]` renvoie `[1;1;2;2;3;3]`.
3. (0.5 point) La fonction `f` ci-dessous est-elle récursive terminale ?

```
let rec f n =
   if n = 0 then 0
   else if n mod 2 = 0 then f (n/2)
   else 1 + f (n/2)
;;
```

   (justifier brièvement).
4. (1 point) écrire une fonction récursive **terminale** `total_len : string list -> int` prenant en argument une liste de chaîne de caractères et renvoyant la somme des longueurs totales de toutes les chaînes. Si vous avez besoin de définir une fonction auxiliaire, vous pouvez choisir de la définir à part ou d'en faire une fonction locale.
5. (1.5 point) écrire une fonction récursive `rotate : 'a list -> 'a list` qui renvoie une copie de la liste placée en argument où le premier élément a été mis à la fin. Par exemple :

   ```
   rotate ["A"; "B"; "C"; "D"]
   ```

   renvoie

   ```
   ["B"; "C"; "D"; "A"]
   ```

   Si l'argument est la liste vide, vous devez renvoyer la liste vide.

3 Problème : gestion d'évènements (10 points)

Attention, dans cet exercice, certaines questions réutilisent des fonctions écrites dans les questions précédentes. Vous pouvez utiliser ces fonctions même si vous n'avez pas réussi à les écrire.

Contexte

On souhaite gérer des listes d'évènements, par exemple comme ceux que l'on peut avoir dans une application de calendrier. Chaque élément possède trois informations :

La date de début : c'est un entier représentant la date de début de l'évènement au format d'heure Unix. Ce dernier est simplement le nombre de secondes écoulées depuis le 1er janvier 1970. C'est une façon commode de représenter le temps, sans se soucier des fuseaux horaires, heures d'été et d'hiver. Tous les langages possèdent des fonctions prédéfinies pour convertir ces entiers en chaînes lisibles, par exemple l'entier 1641403998 correspond au mercredi 5 janvier 2022, 18h 33min 18s, heure de Paris.

La durée de l'évènement exprimée en en secondes

La description de l'évènement exprimée comme une chaîne de caractères

Le but de l'exercice est de modéliser des listes d'évènements et d'écrire des opérations sur ces derniers.

Représentation en OCaml

On se donne les types OCaml suivants :

```
1  type event = {
2    date : int;     (* la date de début au format d'heure Unix *)
3    len : int;      (* la durée, en secondes *)
4    text : string   (* la description, comme une chaîne *)
5    }
6  ;;
7  type calendar = event list ;;
```

Le type `event` représente un évènement et le type `calendar` représente un document comme une liste d'éléments. On dispose de plus d'une fonction `string_of_date : int -> string` qui transforme la date au format Unix en une chaîne de caractères. Par exemple `string_of_date 1641403998` renvoie la chaîne de caractères `"2022/01/05 18:33:18"`.

Questions

Il est vivement recommandé d'utiliser les fonctions prédéfinies sur les listes lorsque c'est possible.

1. (1.5 point) Écrire une fonction `string_of_len : int -> string` qui renvoie une chaîne de caractères représentant la durée passée en argument. Cette dernière est supposée être en secondes. La chaîne doit contenir le nombre d'heures (s'il y en a), de minutes (s'il y en a) et de secondes dans la durée donnée. Par exemple :
   — `string_of_len 10` renvoie `"10s"`
   — `string_of_len 90` renvoie `"1min 30s"`
   — `string_of_len 1000000` renvoie `"277h 46min 40s"` (en particulier on ne convertit pas les heures en jours).
2. (1 point) Écrire une fonction `string_of_event : event -> string` qui renvoie une chaîne de caractères représentant l'évènement. Cette chaîne sera constituée de la concaténation de la chaîne représentant la date (obtenue avec `string_of_date`), de celle de la durée (obtenue avec `string_of_len`) et de celle représentant le texte, séparés par des espaces.
3. (1 point) Écrire une fonction `pr_event : calendar -> unit` qui affiche une valeur de type `calendar` dans la console, un évènement par ligne.
4. (1 point) Écrire une fonction `compare_event : event -> event -> int`. Cette dernière doit renvoyer un entier négatif si le premier évènement commence avant le second, positif s'il commence après le second et nul s'ils commencent à la même date.
5. (1 point) Écrire une fonction `sort_calendar : calendar -> calendar` qui renvoie la copie triée d'un `calendar` ordonnée par date de début d'évènements.
6. (2.5 points) Écrire une fonction `rem_overlap : calendar -> calendar` qui prend un calendrier (une liste d'évènements) et renvoie une copie de ce dernier où les évènements en conflit ont été retirés. Deux évènements sont en conflit si l'un commence avant que l'autre termine. Attention, si deux évènements sont en conflit, il faut retirer l'un des deux (celui que vous souhaitez) mais pas les deux.

   Indication on pourra trier d'abord la liste d'évènements puis la parcourir linéairement en inspectant les évènements deux par deux.
7. (2.5 points) Écrire une fonction `merge_events: calendar -> calendar` prenant en argument un calendrier et qui renvoie un calendrier où les évènements en conflit ayant le même texte sont fusionnés. Par exemple, sur le calendrier :

```
let cal = [
  { start = 1641210000; len = 3600; texte = "Cours L2" } ;
  { start = 1641200000; len = 1800; texte = "Réunion" } ;
```

```
  { start = 1641211200; len = 3600; texte = "Cours L2" } ;
    ]
```

l'expression `merge_event cal` renvoie la valeur

```
[
  { start = 1641200000; len = 1800; texte = "Réunion" } ;
  { start = 1641210000; len = 6000; texte = "Cours L2" } ;
]
```

En effet, dans le premier calendrier, les deux évènements `"Cours L2"` sont en conflit car le premier dure 1h et le second dure aussi 1h mais ne commence que 1200 secondes (20 minutes) après le premier. Les deux fusionnés donnent donc un seul évènement, commençant à la date la plus ancienne des deux, et ayant comme durée totale `3600 + 3600 -1200`. Comme dans la question précédente, il peut être utile, pour avoir un algorithme plus simple, de trier d'abord la liste d'évènements par date de début.

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 :

```
1  let p1 = (10, 12);;
2  let p2 = (-1, false);;
3  let t3 = ("A", "B", 24);;
4
5  let x = fst p1;; (* 10 *)
6  let y = snd p2;; (* false *)
7  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 par une expression de type donnant pour chaque étiquette le type des valeurs associées :

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

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

```
1  let prouge = { x = 0.5; y = 10.3; couleur = "rouge" } ;;
2
3  (* Fonction pour afficher un point coloré *)
4  let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
5         p.x p.y p.couleur;;
```

Un type somme est défini par une expression de type donnant la liste de cas possibles :

```
1  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; 3; 10; 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.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 (* la liste vide est de longueur 0, pair *)
5   | [ _ ] -> false (* la liste a un élément est de longueur 1, imapair *)
6   | _ :: _ :: ll -> liste_long_paire ll (* cas récursif *)
7   ;;
```