Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2015-2016

Examen

lundi 4 janvier 2016

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de **3 heures**.

Les exercices doivent être rédigés **en fonctionnel pur** : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Une fonction écrite en style impératif ne rapportera aucun point.

Les questions sont indépendantes mais il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment (même si elles sont non traitées) ou prédéfinies dans la bibliothèque standard (notamment dans le module sur les listes).

On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre. Cet énoncé a **4 pages**.

Exercice 1 (Expressions). Qu'affiche le toplevel Caml quand on entre l'une après l'autre les expressions suivantes ? Donner pour chaque expression ou définition :

- son type,
- sa valeur,
- et, le cas échéant, son effet de bord,
- ou l'exception levée.

Si l'expression est mal typée ou pas correcte, donner le message d'erreur. Justifier si nécessaire.

1. `let x = "bonne annee "^(string_of_int 2016);;`
2. `'b'^'o'^'f';;`
3. ```
   let head l = match l with
       | [] -> failwith "liste vide"
       | e::tl -> e ;;
   ```
4. `head [];;`
5. ```
   let head l = match l with
       | [] -> print_string "liste vide\n"
       | e::tl -> e ;;
   ```
6. `head [];;`
7. `head [1; 2];;`
8. `let f x = let a = 3 in x + a;;`
9. `let rec g h x = h (h (a+x)) and a = 5;;`
10. ```
    fun x ->
      let id x = x in
      let rec g v m = if m = 0 then [] else v::(g v (m-1)) in
      g id x
    ```
    ```

11. `g f (f a);;`

12. (Bonus)

```
let rec ping x = pong (Some x) and pong (Some y) = ping y;;
```

Exercice 2. Écrire une version du programme suivant qui n'utilise ni boucles, ni références, ni données mutables. La fonction doit avoir exactement le même type et le même comportement.

```
let search_char x file =
  let ci = open_in file in
  try
    begin
      let found = ref false in
      while (not !found) do
        let line = input_line ci in
        found := String.contains line x
      done ;
      !found
    end
  with End_of_file -> false
```

page 21 cours 7

On rappelle que dans le module `String` la fonction `contains` a pour type `string -> char -> bool` et `contains s c` vaut

— `true` si le caractère `c` se trouve dans la chaîne de caractères `s`,
— `false` sinon.

Exercice 3. Le but de cet exercice est de mettre en œuvre une interface polymorphe modélisant les ensembles. On représentera un ensemble par une liste *sans répétitions* :

```
type 'a set = 'a list
```

Par exemple, l'ensemble $S = \{a, b, e\}$ sera représenté par la liste `['a'; 'b'; 'e']`, ou par n'importe quelle autre liste contenant les mêmes éléments. L'utilisation de la récurrence terminale dans les exercices qui suivent sera appréciée.

1. Écrire la fonction `empty : unit -> 'a set` qui renvoie l'ensemble vide, et la fonction `add : 'a -> 'a set -> 'a set` qui ajoute un élément à un ensemble donné. Attention à ne pas créer des répétitions : par exemple, `add 1 (add 1 (empty ()))` doit s'évaluer en `[1]`.
2. Écrire la fonction `inter : 'a set -> 'a set -> 'a set` qui renvoie l'intersection de deux ensembles donnés en entrée. (On rappelle que l'intersection de deux ensembles S et S' est l'ensemble des éléments appartenant à la fois à S et à S').
3. Écrire la fonction `union : 'a set -> 'a set -> 'a set` qui renvoie l'union de deux ensembles donnés en entrée. (On rappelle que l'union de deux ensembles S et S' est l'ensemble des éléments qui appartiennent à S ou à S'. Attention à ne pas produire des répétitions).
4. Écrire la fonction `equal : 'a set -> 'a set -> bool` qui décide si deux ensembles sont égaux (c'est-à-dire, s'ils ont les mêmes éléments) ou non. Par exemple, l'expression `equal [1; 2] [2; 1]` s'évalue en `true`, tandis que `equal [1; 3] [1]` s'évalue en `false`.

5. On rappelle que l'*ensemble quotient* d'un ensemble S par une relation d'équivalence R est l'ensemble des classes d'équivalence de R sur S, c'est-à-dire l'ensemble des sous ensembles non-vides X de S tel que :
   (i) pour tout x, x' appartenant à X, on a $R(x, x')$;
   (ii) pour tout y appartenant à S, s'il existe x appartenant à X tel que $R(x, y)$, alors y appartient à X.
   On représente une relation d'équivalence R comme une fonction `r : 'a * 'a -> bool` telle que : si $R(x, y)$, alors `r x y` s'évalue en `true`, sinon `r x y` s'évalue en `false`.
   Par exemple, la fonction `let three x y = (x mod 3 = y mod 3)` représente l'équivalence modulo 3.
   Écrire la fonction `quotient` de type `('a * 'a -> bool) -> 'a set -> 'a set set`, qui prend en argument une fonction `r`, représentant une relation d'équivalence, et un ensemble `s`, et renvoie l'ensemble quotient de `s` par `r`.
   Par exemple, `quotient three [1; 2; 3; 4; 5]` s'évalue en `[[1; 4]; [2; 5]; [3]]`.

Exercice 4. Nous allons essayer dans cette partie de simuler le comportement des exceptions sans utiliser d'exceptions. Le but est donc de définir un équivalent de la fonction `raise` et de la construction `try ... with`. On ne s'autorisera donc pas l'utilisation de la fonction `raise`, ni de `try`, ni du type `exn` d'OCaml.

1. Écrire un type `'a t` à deux constructeurs, représentant des expressions d'un type `'a` qui peuvent échouer. Autrement dit, une valeur de type `'a t` encapsule :
   — soit une valeur de type `'a` (en cas de réussite)
   — soit un message d'erreur, de type `string` (en cas d'échec)

   Result of 'a
   Failure of string
   exn

2. Écrire les fonctions

   ```
   return : 'a -> 'a t
   fail : string -> 'a t
   ```

   qui construisent une valeur de type `'a t` à partir, respectivement, d'une valeur de type `'a` et d'un message d'erreur. (Les fonctions sont triviales).

3. En utilisant `return` et `fail`, écrire la fonction de recherche dans une liste d'association :

   ```
   find : 'a -> ('a * 'b) list -> 'b t
   ```

   telle que `find k l` renvoie la valeur `v` si le couple `(k,v)` est le premier couple de la liste `l` ayant comme clé `k`. S'il n'existe aucun couple de `l` de clé `k`, alors `find k l` lance un message d'erreur.

4. Écrire une fonction

   ```
   bind : 'a t -> ('a -> 'b t) -> 'b t
   ```

   permettant d'appliquer une fonction à une valeur de type `'a t` en tenant compte des possibilités d'échec. Autrement dit : `bind a f` est l'application de `f` à la valeur de `a` si `a` a une valeur, et échoue si `a` a échoué (avec le même message d'erreur). Indication : observez bien le type de `bind`.

5. Écrire une fonction

   ```
   trywith : 'a t -> (string -> 'a t) -> 'a t
   ```

   qui simule le comportement de la construction `try ... with`. C'est-à-dire : `trywith e f` a la même valeur que `e` si `e` a réussi, et en cas d'échec, le message d'erreur est passé à la fonction `f`. (La fonction `f` simule la partie située après le `with` d'un `try ... with`).

6. À l'aide de `return`, `bind`, `trywith`, `fail` et `find`, écrire une fonction

```
add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list t
```

qui rajoute une entrée à une liste d'association si la clé n'est pas déjà présente, et échoue avec une erreur `"Already_here"` si elle y est déjà.

7. Écrire une fonction

```
map : ('a -> 'b t) -> 'a list -> 'b list t
```

équivalente à `List.map` mais avec notre modèle d'exceptions. Elle doit se comporter exactement comme `List.map`, c'est-à-dire : `map` va appliquer une fonction `f` à tous les éléments d'une liste. Si l'une de ces applications échoue, `map` échoue avec la même erreur et les applications suivantes ne sont pas effectuées.

8. Quel type aurait une fonction équivalente à `List.fold_left` avec ce modèle d'exceptions ? Écrire cette fonction.
9. Quels sont les avantages et/ou les inconvénients de ce modèle d'exceptions par rapport au modèle habituel d'OCaml ?