Examen de Programmation Fonctionnelle

8 Janvier 2008

Durée : 3h

Documents autorisés : cours, TPs, projets.
Livres interdits
Échange de documents interdit — Téléphones portables interdits

Vous devez répondre aux questions en OCaml, en utilisant uniquement un style fonctionnel (exceptions autorisées sauf pour l'exercice 3). Une fonction écrite en style impératif ne rapportera aucun point.

Le barème ci-dessous est indicatif et est susceptible d'être modifié. Les questions sont indépendantes mais il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment.

Exercice 1 (6 points)

Répondez (sans justification) aux questions suivantes :

1. Quelle est la valeur de l'expression suivante :

```
let a = 1000 in
let f =
  let a = 3 in
  fun x -> a + x
in
f 0
```

   Correction : 3

2. On définit :

```
type 'a t = {elts: 'a list}
```

   Quelle est le type de l'expression suivante :

```
fun x -> match x.elts with
| [] -> 0
| a::_ -> a
```

   Correction : `int t -> int`

3. Quelle est le type de l'expression suivante :

```
Some [4]
```

Correction : `int list option`

4. Quelle est le type de l'expression suivante :

```
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
```

Correction : `int -> ('a -> 'a) list`

5. Quelle est le type de la fonction suivante :

```
let f (x, a, b) = if b then a else int_of_float x
```

Correction : `float * int * bool -> int`

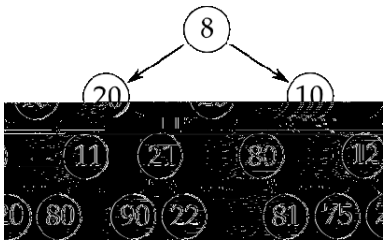
6. Donner la version curryfiée de la fonction précédente.
7. Expliquez les notions d'*égalité structurelle* et *égalité physique*. Quels sont les avantages de l'une ou de l'autre ? Quand doit-on les utiliser ? *(une dizaine de lignes)*

Exercice 2 - Tas (7 points)

Un tas est un arbre binaire, avec une valeur à chaque noeud, répondant à l'unique contrainte suivante :

Chaque noeud (sauf la racine) a une valeur supérieure ou égale à son père.

Attention : Ne confondez pas *tas* et *arbre binaire de recherche*. Exemple : l'arbre suivant est un tas, mais ce n'est pas un arbre binaire de recherche :



On définit le type `tas` de la manière suivante :

```
type tas = F | N of int * tas * tas;;
```

1. Définir une fonction

   ```
   min : tas -> int
   ```

   qui renvoie la plus petite valeur d'un tas. Que proposez-vous de faire dans le cas du tas vide ?
   Correction :

```
exception Empty

let min = function
| F -> raise Empty
| N (a, _ , _) -> a
```

- Moins bien : avec `failwith`
- Pas bien : filtrage non exhaustif

2. Écrire une fonction

   `est_un_tas : tas -> bool`

   qui vérifie si un tas est bien formé.
   Correction :

```
let est_un_tas =
   let rec aux v = function
     | F -> true
     | N (a, t1, t2) -> a >= v && aux a t1 && aux a t2
   in function
     | F -> true
     | N (a, t1, t2) -> aux a t1 && aux a t2
```

3. Définir une fonction

   `ajoute : int -> tas -> tas`

   qui ajoute un élément à un tas. Notez que l'on peut indifféremment insérer le noeud dans le sous-arbre gauche ou droite. Pour éviter d'avoir un arbre déséquilibré, on insérera toujours du même côté, mais on inversera ensuite les deux sous-arbres.
   Correction : Ne pas enlever de point pour les arbres déséquilibrés (énoncé pas clair)

```
let rec ajoute x = function
 | F -> N (x, F, F)
 | N (a, t1, t2) ->
     if x > a then
       N (a, t2, ajoute x t1)
     else
       N (x, t2, ajoute a t1)
```

4. Écrire une fonction

   `fusionne_tas : tas -> tas -> tas`

   qui fusionne deux tas. Indication : si les deux tas sont non-vides, la nouvelle racine est la plus petite des deux racines.
   Correction :

```
let rec fusionne_tas t1 t2 =
  match (t1, t2) with
  | (F, F) -> F
  | (t, F)
  | (F, t) -> t
  | (N (b, t11, t12), N (c, t21, t22)) ->
     if b <= c then
       N (b, fusionne_tas t11 t12, t2)
     else
       N (c, t1, fusionne_tas t21 t22)
```

5. En utilisant la fonction `fusionne_tas`, définir une fonction

   ```
   enleve_min : tas -> tas
   ```

   qui enlève la plus petite valeur d'un tas.
   Correction :

```
let enleve_min = function
 | F -> raise Empty
 | N (a, t1, t2) ->
   fusionne_tas t1 t2
```

6. Écrire une fonction

   ```
   entasse : int list -> tas
   ```

   qui crée un tas à partir des éléments d'une liste
   Correction :

```
let entasse = List.fold_left (fun t e -> ajoute e t) F

(* ou, équivalent : *)
let entasse l =
  let rec aux accu l =
    match l with
     | [] -> accu
     | a::l -> entasse (ajoute a accu) l
  in
  aux F l;;

(* ou, moins bien (mais on n'enlève pas de point) : *)
let rec entasse = function
  | [] -> F
  | a::l -> ajoute a (entasse l);; (* not tail recursive *)
```

7. Écrire une fonction

   ```
   extrais : tas -> int list
   ```

   qui renvoie la liste des éléments du tas, triés par ordre croissant.
   Correction :

```
let rec extrais = function
 | F -> []
 | N (a, _, _) as t -> a::extrais (enleve_min t)
```

8. Déduire des questions précédente une fonction

   ```
   tri : int list -> int list
   ```

   qui trie les éléments d'une liste. Ce tri est connu sous le nom de *tri par tas* (en anglais *heap sort*).
   Correction :

```
let tri l = extrais (entasse l)
```

9. Définir une interface pour un module tas, avec les fonctionnalités que vous venez d'implémenter. Avec votre implémentation et votre interface, est-ce que la fonction `est_un_tas` est utile ? Doit-elle être exportée ? Pourquoi ?

   Correction : Le type doit être abstrait. Il ne doit pas y avoir de moyen de construire des arbres qui ne sont pas des tas.

Exercice 3 - Simulation d'exceptions (7 points)

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 `exception` d'OCaml.

1. Pour cela nous allons considérer que le résultat d'une fonction qui peut échouer est :
   – soit la valeur du résultat (en cas de réussite)
   – soit un message d'erreur, de type `string` (en cas d'échec)
   Définir le type de données `'a t` correspondant.
   Correction :

```
type 'a t = Result of 'a | Exn of string
```

2. Définir les fonctions (triviales)

```
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 (une ligne par fonction).
   Correction :

```
let return a = Result a;;
let fail e = Exn e;;
```

3. Exemple d'utilisation : en utilisant `return` et `fail`, définir la fonction

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

   de recherche dans une liste d'association.
   Correction :

```
let rec find a = function
  | [] -> fail "Not_found"
  | (k, v)::_ when k = a -> return v
  | _::l -> find a l
```

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`.
   Correction :

```
# let bind v f = match v with
  | Exn e -> Exn e
  | Result r -> f r;;
val bind : 'a t -> ('a -> 'b t) -> 'b t = <fun>
```

Petite subtilité (mais pas de point en moins) :

```
# let bind v f = match v with
  | Exn _ -> v
  | Result r -> f r;;
val bind : 'a t -> ('a -> 'a t) -> 'a t = <fun>
```

5. Définir 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`).
   Correction :

```
let trywith v h =
  match v with
   | Exn e -> h e
   | Result r -> Result r
```

6. Application : à l'aide de `return`, `bind`, `trywith`, `fail` et `find`, définir 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.
   Correction :

```
let add k v l =
  trywith
    (bind
       (find k l)
       (fun _ -> fail "Already_here"))
    (function
       | "Not_found" -> return ((k, v)::l)
       | e -> fail e (* Ne pas oublier de rebalancer
                        les autres exceptions *));;
```

7. Définir une fonction

   `iter : ('a -> unit t) -> 'a list -> unit t`

   équivalente à `List.iter` mais avec notre modèle d'exceptions. Elle doit se comporter exactement comme `List.iter`, c'est-à-dire : `iter` va itérer une fonction (produisant un effet de bord, mais qui peut échouer) sur une liste. Si l'un des calculs échoue, alors `iter` échoue avec la même erreur et les calculs suivants ne sont pas effectués.
   Correction :

```
let
```