

# Examen de Programmation Fonctionnelle

1re session — 6 Janvier 2009

Durée : 3h

*Documents autorisés : cours, TPs, projets.*

*Livres interdits — Échange de documents interdit — Ordinateurs et téléphones interdits*

*Vous devez répondre aux questions en OCaml, en utilisant uniquement un style fonctionnel (pas de références, exceptions autorisées). 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 dans le problème il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment.*

*Vous pouvez utiliser les fonctions de la bibliothèque standard, notamment les itérateurs sur les listes.*

## Exercice (8 points)

Répondez (sans justification) aux questions suivantes :

1. Donner le type et la valeur de l'expression suivante :

```
print_int 4
```

**Correction** :-: unit = ()

2. Donner le type et la valeur de l'expression suivante :

```
(fun x y -> x y) (fun x -> x+1) 1
```

**Correction** :-: int = 2

3. Donner le type et la valeur de l'expression suivante :

```
List.fold_left (fun b v -> b + 10 + v) 777 [1; 2]
```

**Correction** :-: int = 800

4. Quel est le type de la valeur suivante ?

```
fun x -> match x with (a, b) -> Some (fun y -> a+b+y)
```

**Correction** :

```
int * int -> (int -> int) option
```

5. On définit :

```
type 'a t1 = {b: 'a; c:t2} and t2 = E of int t1 | F of bool t1 | G
let rec f v =
  v.b +
  match v.c with
  | E w -> f w
  | F w when w.b -> 100
  | _ -> 0
```

Quel est le type de f ? Quel est le type et la valeur de f {b=1; c=E {b=2; c=F {b=true; c=G}}}

**Correction :** f : int t1 -> int  
- : int = 103

6. Recopiez tous les motifs (*patterns*) apparaissant dans la définition suivante (qui utilise le type t1 défini avant) :

```
let g (x, y) =  
  Some (match x with  
    | {b=valeur; c=_} -> (fun z (u, (v, w, t)) -> z + u + v + w))
```

Quel est le type de g ?

**Correction :** Motifs : (x, y), {b=valeur; c=\_}, z et (u, (v, w, t))  
g : 'a t1 \* 'b -> (int -> int \* (int \* int \* 'c) -> int) option

7. Définir un type 'a arbre pour représenter des arbres vérifiant les propriétés suivantes :

- les feuilles sont étiquetées par des fonctions de unit vers 'a;
- la racine est un noeud binaire (deux fils);
- les fils d'un noeud binaire sont des noeuds ternaires (trois fils) ou des feuilles;
- les fils d'un noeud ternaire sont des noeuds binaires.

Écrire une fonction

```
reduit : ('a -> 'a -> 'b) -> ('b -> 'b -> 'b -> 'a) -> 'a arbre -> 'b
```

telle que `reduit f g a` calcule la valeur d'un arbre définie comme suit :

- la valeur d'une feuille est obtenue en appliquant la fonction contenue dans cette feuille;
- la valeur d'un noeud binaire est obtenue en appliquant la fonction f aux valeurs des fils;
- la valeur d'un noeud ternaire est obtenue en appliquant la fonction g aux valeurs des fils.

**Correction :**

```
type 'a arbre = A2 of ('a a3 * 'a a3)  
and 'a a3 = A3 of ('a arbre * 'a arbre * 'a arbre) | F of (unit -> 'a)
```

```
let rec reduit f g (A2 (a, b)) =  
  f (parcours3 f g a) (parcours3 f g b)  
and parcours3 f g = function  
| F f -> f ()  
| A3 (a, b, c) ->  
  g (reduit f g a) (reduit f g b) (reduit f g c);;
```

8. On définit

```
let etoiles n =  
  let compt = ref 0 in  
  for i = 1 to n do  
    for j = 1 to i do  
      print_string "*";  
      compt := !compt + 1  
    done;  
    print_newline ();  
  done;  
  print_int (!compt);  
  print_string " etoiles";;
```

Qu'affiche `etoiles 4` ?

Réécrire la fonction `etoiles` sans utiliser ni références, ni données mutables, ni boucles. La fonction doit avoir exactement le même type et le même comportement.

**Correction :**

```

*
**
***
****
10 etoiles

let etoiles n =
  let rec ligne acc nb =
    if nb = 0 then acc else (print_string "*"; ligne (1+acc) (nb -1))
  in
  let rec aux acc nb =
    if nb > n
    then acc
    else (let n = ligne acc nb in print_newline (); aux n (nb+1))
  in
  print_int (aux 0 1);
  print_string " etoiles";

```

Ou, moins bien (pas récursive terminale) :

```

let etoiles n =
  let rec ligne nb =
    if nb = 0 then 0 else (print_string "*"; 1+ligne (nb -1))
  in
  let rec aux nb =
    if nb > n
    then 0
    else (let n = ligne nb in print_newline (); n + aux (nb+1))
  in
  print_int (aux 1);
  print_string " etoiles"

```

9. Qu'est-ce que la *curryfication* (en une dizaine de lignes)? Donnez un exemple.

## Problème - (12 points)

Le but du problème est de réaliser un outil d'aide à la création d'emplois du temps pour une semaine de cours dans une école.

### 1 Horaires

Nous allons définir un module `Horaires` pour manipuler les moments de la semaine (dates). Ce module nous permettra de convertir un moment de la semaine donné par un triplet (jour, heures, minutes) en une représentation interne des dates (et vice-versa). Il permettra aussi de faire des calculs sur les dates. Nous utiliserons comme représentation interne des dates le nombre de secondes depuis le lundi matin 0h00. L'utilisateur du module ne doit pas avoir connaissance de la représentation interne (ce qui permettra éventuellement de la changer plus tard pour par exemple utiliser des flottants).

1. Définir un type pour représenter un jour de la semaine (lundi, mardi, etc).
2. Définir un type pour représenter un moment de la semaine à l'aide d'un jour de la semaine, une heure et un nombre de minutes.
3. Définir un type `date` pour la représentation interne des dates.
4. Écrire l'interface (signature) du module de gestion des dates, comportant notamment les valeurs suivantes :
  - une valeur `borne_inf` de type `date` correspondant à la date du début de la semaine (borne inférieure de l'intervalle de dates correspondant à une semaine). Avec la représentation interne proposée, c'est juste la valeur 0.

- une valeur borne\_sup correspondant à la borne supérieure de la semaine. On considérera une semaine comme un intervalle de dates fermé à gauche, ouvert à droite, donc cette borne supérieure ne fait pas partie de la semaine.
- une fonction date\_of\_dhm qui convertit en date un moment de la semaine donné par un jour, heure et minute;
- une fonction dhm\_of\_date qui fait la conversion inverse.

5. Écrire l'implémentation du module Horaire.

**Correction :**

```

module Horaire =
  (struct
    type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
    type date = int
    let nbsecondessemaine = 86400 * 7

    let date_of_hm h m = (h*3600 + m*60)

    let date_of_dhm j h m =
      let jj = 86400 * (match j with
        | Lundi -> 0
        | Mardi -> 1
        | Mercredi -> 2
        | Jeudi -> 3
        | Vendredi -> 4
        | Samedi -> 5
        | Dimanche -> 6)
      in (jj + date_of_hm h m) mod nbsecondessemaine

    let dhm_of_date s =
      let jj = (s / 86400) in
      let j = match jj mod 7 with
        | 0 -> Lundi
        | 1 -> Mardi
        | 2 -> Mercredi
        | 3 -> Jeudi
        | 4 -> Vendredi
        | 5 -> Samedi
        | 6 -> Dimanche
        | _ -> assert false
      in
      let sj = s - (jj * 86400) in
      let h = sj / 3600 in
      let m = (sj mod 3600) / 60 in
      (j, h, m)

    let borne_sup = nbsecondessemaine

    let borne_inf = 0

    let sub_min d m = d - (m * 60)
  end :
  sig
    type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
    type date (* doit etre abstrait *)

    val date_of_dhm : jour -> int -> int -> date
    val dhm_of_date : date -> jour * int * int
    val borne_inf : date
    val borne_sup : date
  end

```

```

    val sub_min : date -> int -> date (** may return negative date *)
  end)

```

Vous pourrez éventuellement plus tard être amenés à ajouter des fonctions dans ce module. Vous préciserez à chaque fois les lignes à ajouter dans l'interface et dans l'implémentation.

## 2 Créneaux et plannings

Un créneau est un intervalle défini par sa date de début et de fin (de type `Horaire.date`). Sauf cas particuliers, on considérera ces intervalles fermés à gauche et ouverts à droite. Un planning est un ensemble de créneaux. On définit les types `creneau` et `planning` de la façon suivante :

```

type creneau = {debut : Horaire.date;
                 fin   : Horaire.date}
type planning = creneau list

```

Pour l'instant on supposera que la date de fin est toujours supérieure ou égale à la date de début.

*Rappel* : Vous pouvez utiliser les fonctions de comparaisons standards de OCaml même pour un type abstrait (`<`, `<=`, `max`, `compare`, etc.).

1. Écrire une fonction `date_libre` de type `Horaire.date -> planning -> bool` qui renvoie `true` si la date n'est dans aucun créneau du planning.

**Correction :**

```

let rec date_libre date = function
  | [] -> true
  | c::p -> (date < c.debut || date >= c.fin) && date_libre date p

let date_libre date p =
  List.for_all (fun c -> date < c.debut || date >= c.fin) p

```

2. Écrire une fonction `trie_planning` qui trie un planning selon la date de début des créneau, et, si les dates de début sont égales, selon la date de fin. Vous utiliserez la fonction `List.sort` de la bibliothèque standard, dont voici la documentation en français :

```

val sort : ('a -> 'a -> int) -> 'a list -> 'a list

```

*Trié une liste dans l'ordre croissant pour un fonction de comparaison donnée. Cette fonction de comparaison doit retourner 0 si ses arguments sont égaux, un entier strictement positif si le premier argument est plus grand, et un entier strictement négatif si le premier argument est plus petit.*

**Correction :**

```

let trie_planning = List.sort (fun a b ->
  let v = compare a.debut b.debut in
  if v = 0 then compare a.fin b.fin
  else v)

```

3. Écrire une fonction

```

  fusion_plannings : planning list -> planning

```

qui fusionne les plannings d'une liste, c'est-à-dire qui construit un planning contenant les créneaux de tous les plannings de la liste.

**Correction :**

```

let rec fusion_plannings = function
  | [] -> []
  | p::l -> p@(fusion_plannings l)

let fusion_plannings = List.fold_left (fun res p -> res@p) [] (* bof *)

let fusion_plannings pl = List.fold_right (0) pl []
  (* plus efficace mais pas tail rec, equivalente a la premiere des 3 *)

```

4. On appellera *planning canonique* un planning qui vérifie les conditions suivantes :

- Il est trié;
- les créneaux sont disjoints deux à deux (pas de chevauchement);
- la borne supérieure d'un créneau n'est pas égale à la borne inférieure du créneau suivant.

La *projection* d'un planning  $p$  est le planning canonique  $p'$  qui vérifie la propriété suivante : une date est dans un créneau de  $p'$  si et seulement si elle est dans (au moins) un créneau de  $p$ .

Quelle est la projection du planning dont les créneaux sont les suivants ?

- lundi 8h30 à 10h30
- lundi 9h30 à 11h30
- lundi 11h30 à 13h30
- lundi 14h30 à 16h30

Écrire une fonction `projection_planning` qui calcule la projection d'un planning.

**Correction :**

```

let projection_planning p =
  let rec aux = function
    | [] -> []
    | [c] -> [c]
    | c1::c2::l ->
      if c1.fin < c2.debut (* strict *)
      then c1::(aux (c2::l))
      else aux ({debut=c1.debut; fin=(max c1.fin c2.fin)}::l)
  in aux (trie_planning p)

```

5. Écrire une fonction `complementaire_planning` qui calcule le planning canonique  $p'$  complémentaire d'un planning  $p$  donné. C'est-à-dire : une date est dans un créneau de  $p'$  si et seulement si elle n'est dans aucun créneau de  $p$ .

**Correction :**

```

let complementaire_planning p =
  let rec aux h = function
    | [] ->
      if h >= Horaire.borne_sup
      then []
      else [{debut=h; fin=Horaire.borne_sup}]
    | a::l ->
      if h < a.debut
      then {debut=h; fin=a.debut}::(aux a.fin l)
      else (aux a.fin l)
  in
  aux Horaire.borne_inf (projection_planning p)

```

### 3 Aide à la création d'emplois du temps

Dans cette partie, nous allons écrire des fonctions pour aider au placement d'un nouveau cours sur un ou des plannings.

1. Écrire une fonction

```
possibilites_debut_creneau : int -> planning -> planning
```

telle que `possibilites_debut_creneau m p` renvoie le planning canonique dont les créneaux représentent les possibilités pour placer le début d'un cours de  $m$  minutes sur le planning  $p$  sans que le cours chevauche un créneau existant. (Dans le planning résultat, les créneaux seront exceptionnellement fermés à gauche et à droite). Vous pouvez avoir besoin de rajouter une ou des fonctions dans le module `Horaire`. On supposera pour simplifier qu'il n'est pas possible de placer un cours à cheval sur deux semaines.

**Correction :**

```
let possibilites_debut_creneau m p =
  let rec reduit = fonction
    | [] -> []
    | c::l ->
      let l' = reduit l in
      let nouvfin = Horaire.sub_min c.fin m in
      if nouvfin >= c.debut (* le egal est important *)
      then {debut=c.debut; fin=nouvfin}::l'
      else l'
  in
  reduit (complementaire_planning p)
```

- Parfois, on souhaite placer un cours sur un planning quelconque parmi plusieurs, par exemple lorsqu'il s'agit de plannings de salles qui sont équivalentes. Écrire une fonction `possibilites_debut_liste` qui prend un entier  $m$  et une liste de plannings, et renvoie le planning canonique dont les créneaux représentent les possibilités pour placer le début d'un cours de  $m$  minutes sur un des plannings de la liste. (créneaux fermés également)

**Correction :**

```
let possibilites_debut_liste m pl =
  projection_planning
  (fusion_plannings (List.map (possibilites_debut_creneau m) pl))
```

- La fonction `complementaire_planning` fonctionne également pour des plannings avec créneaux fermés (sauf qu'il faut considérer la semaine comme intervalle ouvert). Mais elle ne fonctionne pas pour des plannings avec créneaux ouverts. Donnez un exemple. Écrivez une nouvelle fonction `complementaire_canonique_ouverts` qui calcule le complémentaire d'un plannings canonique dans lesquels les créneaux sont ouverts à droite et à gauche.

**Correction :**

```
let complementaire_canonique_ouverts p =
  let rec aux h = fonction
    | [] ->
      if h >= Horaire.borne_sup
      then []
      else [{debut=h; fin=Horaire.borne_sup}]
    | a::l ->
      if (a.debut <> Horaire.borne_inf) && (h <= a.debut)
      then {debut=h; fin=a.debut}::(aux a.fin l)
      else (aux a.fin l)
  in
  aux Horaire.borne_inf p
```

- En déduire une fonction

```
intersection_fermes : planning list -> planning
```

qui calcule l'intersection  $p'$  des créneaux des plannings d'une liste de plannings avec créneaux fermés. C'est-à-dire : une date est dans un créneau de  $p'$  si et seulement si elle est dans au moins un créneau de chacun des plannings.

**Correction :**

```

let intersection_fermes pl =
  complementaire_ouverts_canonique
    (fusion_plannings (List.map complementaire_planning pl))

```

5. Parfois, on souhaite placer un créneau sur plusieurs plannings (par exemple le planning de l'enseignant, du groupe d'élèves et de la salle). Quelle fonction utiliser pour calculer les possibilités pour placer le début d'un cours étant données les possibilités pour le début de cours de chacun des plannings sur lesquels on veut le placer. Attention : chacun des créneaux des plannings de la liste en paramètre est un créneau fermé.

**Correction :**

```

intersection_fermes

```

## 4 Amélioration du typage

### 4.1 Créneaux

On a supposé jusque là que les créneaux étaient tous bien formés, c'est-à-dire que la date de fin était supérieure ou égale à la date de début. Écrire un module (interface et implémentation) permettant de manipuler des créneaux bien formés. La vérification est faite dynamiquement au moment de la création du créneau. Il ne doit pas y avoir de possibilité de créer un créneau mal formé. De quelles fonctions avez-vous besoin dans l'interface ?

**Correction :** Type `creneau` abstrait + fonction de construction et destructeurs.

### 4.2 Plannings

Nous avons vu dans la section précédente qu'il faut être très vigilant sur le type de créneaux manipulés. Nous souhaitons éviter ces erreurs en ajoutant dans le type des plannings l'information sur la forme des créneaux manipulés. Les trois sortes de plannings que nous avons vus sont les suivantes :

- planning avec des créneaux fermés à gauche et ouverts à droite ;
- planning avec des créneaux fermés à gauche et à droite ;
- planning avec des créneaux ouverts à gauche et à droite.

Nous allons modifier le type `planning` pour lui rajouter un paramètre :

```

type 'a planning = creneau list

```

Notez que ce paramètre n'intervient pas dans la définition du type. Il servira juste à préciser dans l'interface du module la forme des planning que nous manipulons. La valeur de ce paramètre de type utilisera des types variants polymorphes avec les constructeurs suivants :

- 'Ouverts\_droite pour désigner des intervalles ouverts à droite fermés à gauche ;
- 'Fermes pour désigner des intervalles fermés ;
- 'Ouverts pour désigner des intervalles ouverts.

Pour être sûrs d'utiliser les bons types, nous allons cacher l'implémentation des plannings en abstrayant ce nouveau type `planning`. Nous allons donc créer un module `Plannings`, qui contiendra les valeurs suivantes :

- `planning_vide` (un planning vide avec créneaux ouverts à droite)
- `planning_vide_ouverts` (un planning vide avec créneaux ouverts)
- `planning_vide_fermes` (un planning vide avec créneaux fermés)
- `ajoute_creneau` qui rajoute un créneau à un planning quelconque
- les fonctions `fusion_plannings`, `complementaire_plannings`, `complementaire_ouverts_canonique`
- une valeur `complementaire_fermes`, définie par
 

```

let complementaire_fermes = complementaire_planning

```
- la fonction `projection_plannings`, valable pour les plannings à intervalles fermés ou ouverts à droite, mais pas pour les plannings à intervalles ouverts.

1. Écrire l'interface du module `Plannings`

**Correction :**

sig

```
type 'a planning
val planning_ouvert : [ 'Ouverts_droite ] planning
val planning_ouvert_ferme : [ 'Fermes ] planning
val planning_ouvert_ouvert : [ 'Ouverts ] planning
val ajoute_creneau :
  Semaine.date -> Semaine.date -> 'a planning -> 'a planning
val trie_planning : 'a planning -> 'a planning
val to_list : 'a planning -> creneau list (* pour eviter ca,
                                         il faut utiliser les types
                                         prives *)

val fusion_plannings : 'a planning list -> 'a planning
val projection_planning :
  ([< 'Ouverts_droite | 'Fermes ] as 'a) planning -> 'a planning
val complementaire_planning :
  [ 'Ouverts_droite ] planning -> [ 'Ouverts_droite ] planning
val complementaire_ferme :
  [ 'Fermes ] planning -> [ 'Ouverts ] planning
val complementaire_ouvert_canonique :
  [ 'Ouverts ] planning -> [ 'Fermes ] planning
```

end

2. Proposez une idée similaire pour distinguer les plannings canoniques des autres. Écrire la nouvelle interface du module Plannings.