

## Examen du 19 mai 2008

Les notes de TD manuscrites de cette année sont les seuls documents autorisés.

Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse. Le barème est indicatif.

**Important.** Les types et l'utilisation des fonctions de la bibliothèque standard d'Ocaml mentionnées (ou pouvant être utiles) dans les questions suivantes sont donnés à la fin de ce document.

### 1 Typage (3 points)

**Question 1.1** Donner le type de `r` après chaque déclaration.

```
let r = ref ([], [])  
let _ = r := (snd !r , fst !r)
```

**Question 1.2** Donner le type de la fonction `f` suivante :

```
let f x = if x then raise Not_found else raise Exit
```

**Question 1.3** On donne les deux fonctions suivantes :

```
let foo x = x.a := (snd x.b) + 1  
let bar x = x.c <- [(fst x.b)]::x.c
```

Définir un type `'a t` pour que l'on ait :

```
val foo : 'a t -> unit = <fun>  
val bar : 'a t -> unit = <fun>
```

### 2 Évaluation (4 points)

**Question 2.1** Donner (en le justifiant) les valeurs des variables `a` et `b` après la dernière expression.

```
let make_ref x = ref x  
let a = ref 3  
let b = make_ref a  
let _ = !b := !a+1
```

**Question 2.2** Donner (en le justifiant) les valeurs de chaque case du tableau `m` après la dernière expression.

```

let m = Array.make 4 (Array.make 2 0)
let _ =
  for i = 0 to 3 do
    for j = 0 to 1 do
      m.(i).(j) <- i
    done;
  done

```

### 3 Compilation séparée (3 points)

On considère les extraits suivants des fichiers `fa.mli`, `fa.ml`, `fb.mli`, `fc.ml` et `fp.ml`. Les points de suspension `...` représentent du code qui ne dépend pas d'autres modules.

- `fa.mli`

```

type t
val make : string -> t
val extract : t -> int

```

- `fa.ml`

```

type t = { a: string ; b: Fc.t }
let make s = { a = s ; b = Fc.init s }
let extract { b = (_,i) } = i

```

- `fb.mli`

```

{ let atoi s = int_of_string s }

rule automate = parse
  ['0' - '9']+ { Fa.make (Lexing.lexeme lexbuf) }
  | ...

{ let analyse v = automate (Lexing.from_string v) }

```

- `fc.ml`

```

type t = string * int
let init s = (s,Fb.atoi s)

```

- `fp.ml`

```

let foo s = Fa.extract (Fb.analyse s) + 100

```

**Question 3.1** Dessiner le graphe de dépendance entre les unités de compilation `Fa`, `Fb`, `Fc` et `Fp`.

**Question 3.2** Est-il possible de compiler les modules précédents ? Si oui, donner – dans l'ordre – les lignes de compilation (et édition de liens) `ocamllex` et `ocamlc` permettant de le faire. Si non, expliquer pourquoi.

## 4 Programmation : Classes disjointes (10 points)

Le but de cette partie est de développer une structure de données pour représenter une partition d'un ensemble fini d'entiers. Le type abstrait de cette structure de données est défini par la signature suivante :

```
module type PARTITION = sig
  type t
  type intset
  val creer : int -> t
  val fusion : int -> int -> t -> unit
  val classe_de : int -> t -> intset
end
```

Cette signature contient deux définitions de type : le type `t` est celui de la partition et le type `intset` représente un ensemble d'entiers. La fonction `creer` permet de créer une partition *initiale* de l'ensemble  $\{0, 1, \dots, n - 1\}$  (où  $n$  est l'entier passé en argument à cette fonction) telle que chaque entier est l'unique élément de sa classe. La fonction `fusion` permet de fusionner les classes de deux entiers (après cette opération, les classes de ces deux entiers sont donc identiques). Enfin, la fonction `classe_de` retourne la classe d'un entier (*i.e.* un ensemble d'entiers).

Afin de ne pas dépendre d'une implémentation particulière des ensembles d'entiers, nous allons implémenter cette structure de donnée comme un foncteur paramétré par un module de signature `INTSET` définie de la manière suivante :

```
module type INTSET = sig
  type t
  val singleton : int -> t
  val union : t -> t -> t
  val iter : (int -> unit) -> t -> unit
end
```

Cette signature définit un type abstrait `t` pour les ensembles finis d'entiers. Un appel `singleton i` crée un ensemble contenant un unique entier `i`. L'union de deux ensembles `a` et `b` est réalisée par l'appel `union a b`. Enfin la fonction `iter` permet d'itérer une fonction de type `int -> unit` sur un ensemble d'entiers. La définition du foncteur `Partition` est donc la suivante :

```
module Partition( S : INTSET ) : PARTITION = struct
  ...
end
```

**Question 4.1** En utilisant un tableau d'ensembles d'entiers pour représenter une partition, donner les définitions des types `t` et `intset` du foncteur `Partition`.

**Question 4.2** À l'aide de ces deux types (et du paramètre `S`), donner la définition des fonctions `creer`, `fusion` et `classe_de`.

**Question 4.3** Quelle contrainte faut-il ajouter à la définition du foncteur `Partition` pour que les valeurs de type `intset` de la signature `PARTITION` puissent être utilisées comme des ensembles d'entiers de type `S.t` ?

**Question 4.4** En utilisant les fonctions logiques sur les *bits*, définir une implémentation d'un module `I` réalisant la signature `INTSET` pour des ensembles d'entiers pouvant contenir au plus les entiers de 0 à 29 (les entiers positifs en `Ocaml` étant codés sur 30 bits).

**Question 4.5** Donner les expressions pour :

- créer un module `P` réalisant la signature `PARTITION` à l'aide du foncteur `Partition` et du module `I` défini dans la question précédente ;
- créer une partition initiale pouvant contenir jusqu'à 8 entiers (de 0 à 7) ;
- joindre les classes des entiers 1 et 2 ;
- joindre les classes des entiers 4 et 6 ;
- afficher – sans redondance – toutes les classes de la partition.

## Rappels.

`fst : ('a * 'b) -> 'a`

retourne la composante gauche d'une paire.

`snd : ('a * 'b) -> 'b`

retourne la composante droite d'une paire.

`lsl : int -> int -> int`

`int i j` retourne l'entier obtenu en décalant `i` vers *la gauche* de `j` bits.

`lor : int -> int -> int`

`lor i j` retourne l'entier résultant du *ou logique* bit à bit entre les entiers `i` et `j`

`land : int -> int -> int`

`land i j` retourne l'entier résultant du *et logique* bit à bit entre les entiers `i` et `j`

`Array.length : 'a array -> int`

`Array.length t` retourne la longueur du tableau `t`.

`Array.init : int -> (int -> 'a) -> 'a array`

`Array.init n f` retourne un nouveau tableau de longueur `n`, où chaque élément d'indice `i` est initialisé avec le résultat de `f i`.

`Array.make : int -> 'a -> 'a array`

`Array.make n v` retourne un nouveau tableau de longueur `n`, où chaque élément est initialisé avec la valeur `v`.