Licence STS, semestre 4 — 2009–2010
Langages et Génie Logiciel (Info 223) — 27 mai 2010
`http://www.lri.fr/~paulin/LGL`

Examen Session 1

L'examen est composé de 7 pages. La durée est de 2 heures. Le sujet est long mais sera noté sur un barême supérieur à 20. Les notes de TD manuscrites ainsi que les fiches de cours de cette année sont les seuls documents autorisés à l'examen.

1 Typage et évaluation d'expressions Ocaml (5 points)

1. On introduit un type `r`. Dire si les fonctions `f` et `g` sont bien typées et si oui donner leur type :

```
type r = { mutable p : int; b : float ref}
let f x = x.p <- int_of_float !(x.b); x.b := float_of_int x.p
let rec g a b = if a < !b then raise Exit
                else if a = !b then []
                else (b:= !b+1; a::(g (a-1) b))
```

Correction :

```
val f : r -> unit = <fun>
val g : int -> int ref -> int list = <fun>
```

2. On introduit les fonctions suivantes :

```
let f a x = if !a < x then a:= !a+1; !a
let g1 x = let a = ref 0 in f a x
let g2 = let a = ref 0 in fun x -> f a x
```

Donner le type de ces fonctions et le résultat de l'évaluation des expressions suivantes.

```
g1 1;;
g2 1;;
g1 0;;
g2 0;;
g1 2;;
g2 2;;
```

Correction :

```
val f : int ref -> int -> int = <fun>
val g1 : int -> int = <fun>
val g2 : int -> int = <fun>
g1 1;; (* 1 *)
g2 1;; (* 1 *)
g1 0;; (* 0 *)
g2 0;; (* 1 *)
g1 2;; (* 1 *)
g2 2;; (* 2 *)
```

2 Tests (7 points)

Le programme suivant calcule x^n avec x un entier et n un entier positif en utilisant une méthode dichotomique.

```
let exp x n =
  let rec iter m y n =
    if n = 0 then y
    else
      let n' = n/2 and m' = m * m in
      let y' = if n mod 2 = 0 then y else y * m in
      iter m' y' n'
  in iter x 1 n
```

1. Justifier la terminaison de la fonction `iter`.

   Correction : *Lors de chaque appel récursif de la fonction* `iter`*, l'argument* n *décroit strictement* $n/2 < n$ *si* $n > 0$ *mais reste supérieur ou égal à* 0.

2. Proposer une fonction `test_exp` qui étant donnés x, n et un entier r, vérifie que le résultat de `exp` x n est égal à r et échoue sinon.

   Correction :

```
exception TestError
let test_exp x n r =
  let r' = exp x n in
    if r' <> r then
      begin
        Printf.printf "exp %d %d = %d - résultat attendu: %d" x n r' r;
        raise TestError
      end
```

3. Proposer 5 tests correspondant à des valeurs remarquables de la fonction `exp`.

   Correction : *On teste pour des valeurs de* x *et de* n *qui valent* 0 *ou* 1.

```
let _ = test_exp 0 4 0
let _ = test_exp 5 0 1
let _ = test_exp 0 0 1
let _ = test_exp 1 5 1
let _ = test_exp 4 1 4
```

4. Ecrire une fonction `test_exp_prop` qui prend en argument x et n et vérifie que la fonction `exp` vérifie les égalités suivantes :

$$x^{2n} = x^n \times x^n \qquad\qquad x^{2n+1} = x \times x^n \times x^n$$

   Correction :

```
let test_exp_prop x n =
  let rn = exp x n and r2n = exp x (2*n) and r2np1 = exp x (2*n+1)
  in let rn2 = rn * rn in
    if r2n <> rn2 then
      begin
        Printf.printf "exp %d %d = %d <> (%d)^2\n" x (2*n) r2n rn2;
```

```
      raise TestError
    end;
  if r2np1 <> rn2 * x then
    begin
      Printf.printf "exp %d %d = %d <> (%d)^2 * %d \n" x (2*n+1) r2np1 rn2
      raise TestError
    end
```

5. Ecrire une fonction qui étant donnés des entiers $max, maxn, k$, teste la fonction `exp` en utilisant `test_exp_prop` sur k entrées choisies aléatoirement avec x compris entre 0 et max et n entre 0 et $maxn$.

   Correction :

```
let test_random max maxn k =
  for i = 1 to k do
    let x = Random.int (max+1) and n = Random.int (maxn+1) in
      test_exp_prop x n
  done
```

3 Modules et programmation (11 points)

Le but de cet exercice est d'écrire un programme qui résoud le jeu du Sudoku.

On rappelle que le jeu de Sudoku consiste à compléter un tableau de 9 lignes sur 9 colonnes par des chiffres allant de 1 à 9 de manière à ce que chaque chiffre apparaisse une et une seule fois sur chaque ligne, sur chaque colonne et dans chacun des 9 carrés de 3 lignes sur 3 colonnes qui constituent le tableau.

jeu initial

3				**5**	**4**			**6**
7			**9**				**3**	**5**
					1	**4**		**9**
		3				**5**	**2**	
			1		**5**			
	1	**7**				**6**		
1		**5**	**8**					
6	**7**				**9**			**4**
9			**3**	**6**				**7**

jeu complété

3	2	9	7	**5**	**4**	8	1	**6**
7	4	1	**9**	8	6	2	**3**	**5**
8	5	6	2	3	**1**	**4**	7	**9**
4	9	**3**	6	7	8	**5**	**2**	1
2	6	8	**1**	9	**5**	7	4	3
5	**1**	**7**	4	2	3	**6**	9	8
1	3	**5**	**8**	4	7	9	6	2
6	**7**	2	5	1	**9**	3	8	**4**
9	8	4	**3**	**6**	2	1	5	**7**

Pour résoudre le jeu on utilisera un module pour représenter le jeu (structure de `map`) et un module pour représenter des ensembles d'entiers qui serviront à modéliser l'espace de recherche de solutions.

La signature du module pour représenter les ensembles d'entiers est la suivante :

```
module type INTSET =
sig
  type t (* le type des ensembles d'entiers *)
  val empty : t (* ensemble vide *)
  val is_empty : t -> bool (* teste si un ensemble est vide *)
  val mem : int -> t -> bool (* teste la présence d'un entier *)
  val add : int -> t -> t (* ajoute un entier à un ensemble *)
  val remove : int -> t -> t (* supprime un entier d'un ensemble *)
```

```
  val filter : (int -> bool) -> t -> t
    (* crée un nouvel ensemble ne contenant que les éléments qui vérifient
       le prédicat passé en argument *)
  val choose : t -> int (* renvoie un élément quelconque d'un ensemble *)
  val cardinal : t -> int (* nombre d'éléments d'un ensemble *)
end
```

Pour représenter le jeu lui-même on utilise une structure de `map` qui à un couple d'entiers (i, j) représentant la case sur la i-ème ligne et la j-ème colonne associe la valeur stockée dans cette case. On ne stocke que les cases non-vides. La signature du module correspondant est la suivante :

```
module type BOARD =
sig
  type t
  val empty : t (* map vide *)
  val add : int*int -> int -> t -> t
    (* ajoute un entier associé à un couple de coordonnées *)
  val mem : int*int -> t -> bool
    (* teste la présence d'une entrée correspondant à un couple de
       coordonnées *)
  val find : int*int -> t -> int
    (* renvoie l'entier associé à une position
       lance l'exception Not_found s'il n'y a pas d'entier associé *)
  val fold : (int*int -> int -> 'a -> 'a) -> t -> 'a -> 'a
end
```

Cette signature contient la définition du type `t` qui représente des *maps* qui associent un entier à un couple d'entiers correspondant à une position sur le tableau du jeu. Les lignes et colonnes sont numérotées de 1 à 9. La fonction `fold` permet de calculer une donnée de type $'a$ en traitant successivement tous les éléments d'une map. Si on suppose que la map m contient l'ensemble d'associations suivant : $\{(1,1) \mapsto 4, (1,2) \mapsto 6, (3,3) \mapsto 9\}$ alors `fold` $f\, m\, x$ calcule : $f\,(3,3)\,9\,(f\,(1,2)\,6\,(f\,(1,1)\,4\,x))$. Par exemple la liste des positions occupées dans une map se calcule simplement par l'expression : (`fold` (**fun** $(i,j)\ n\ l \rightarrow (i,j) :: l$) m []).

On suppose que l'on a implanté un module `IntSet` dont la signature est `INTSET` et un module `Board` dont la signature est `BOARD`. On pourra par exemple utiliser l'expression `Board.empty` qui est de type `Board.t` et représente un jeu vide.

Pour rechercher les solutions possibles du jeu on introduit une nouvelle structure de données que l'on appellera une *configuration*. Une *configuration* est une liste formée de triplets (n, i, s) avec n et i des entiers et s un ensemble d'entiers (de type `IntSet.t`). Un triplet (n, i, s) correspond au fait que l'entrée n sur la ligne i peut se trouver sur une des colonnes $j \in s$. Une configuration est *compatible* avec un jeu b si elle contient les entrées (n, i, s) pour lesquelles n n'a pas encore été positionné sur la i-ème ligne du jeu et si les colonnes proposées ne sont pas en conflit direct (même colonne ou carré) avec d'autres positions de n dans le jeu.

Ainsi dans le jeu précédent, la configuration contiendra l'entrée $(2, 1, \{2, 3, 4, 7\})$ car l'entier 2 peut être positionné sur la première ligne seulement sur les colonnes $2, 3, 4, 7$. Il ne peut pas être en colonne 1 car celle-ci est déjà occupée par 3 et ne peut pas être sur la colonne 8 car il y a déjà un 2 dans cette colonne. La configuration ne contiendra pas d'entrée $(3, 1, s)$ car l'entier 3 est déjà positionné dans le jeu sur la 1ère ligne. Si un entier n sur une ligne i est associé à l'ensemble vide alors c'est que le jeu n'a pas de solution. On introduit :

```
type configuration = (int * int * IntSet.t) list
```

Les questions suivantes permettent de construire progressivement la solution du problème. On pourra utiliser les opérations demandées dans une question pour résoudre les questions suivantes. On pensera également à utiliser les opérateurs sur les listes comme `List.map`, `List.sort`...

1. Construire un objet `configuration0` de type `configuration` qui correspond à un jeu vide, c'est-à-dire qui contient tous les triplets (n, i, s) pour $n, i \in \{1, \ldots, 9\}$ et $s = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. On suggère d'introduire la liste `digits` $= [1; 2; 3; 4; 5; 6; 7; 8; 9]$.

   Correction :

```
let digits = [1;2;3;4;5;6;7;8;9]
let all_set = List.fold_right IntSet.add digits IntSet.empty
let configuration0 =
  let all_lines = List.map (fun j -> j, all_set) digits in
  let all_vals =
   List.map (fun n -> List.map (fun (j,s) -> (n,j,s)) all_lines)
            digits
  in List.flatten all_vals
```

2. Construire une fonction `update_configuration` qui prend en entrée une position (i, j), un entier n, et une configuration *conf* et qui renvoie une nouvelle configuration en retirant tous les choix incompatibles avec l'ajout de n à la position (i, j) dans le jeu. Pour cela on considère chaque triplet (m, l, s) de *conf* et on met éventuellement à jour l'ensemble de colonnes s en retirant les colonnes incompatible avec le fait de mettre l'entier n à la position (i, j) dans le jeu. C'est-à-dire :
   - si $l = i$ alors retirer j de s (si n est en position (i, j) alors il n'y a pas d'autre entier à cette position) ;
   - si $m = n$ alors on retire c de s soit si $c = j$ (l'entier est dans la même colonne) soit si (l, c) et (i, j) sont dans le même carré 3×3.
     On remarquera que si on numérote les carrés 3×3 de gauche à droite et de haut en bas alors le carré de la case (i, j) a pour numéro $3 * ((i - 1)/3) + (j - 1)/3 + 1$.

   Correction :

```
let square i j = 3*((i-1)/3) + (j-1)/3 + 1
(** ordre sur les éléments de la configuration *)
let conf_compare (_,_,s1) (_,_,s2)
   = compare (IntSet.cardinal s1) (IntSet.cardinal s2)
(** k est le carré associé à (i,j) ce prédicat est vrai si (i,j)
    et (i1,j1) ne sont pas sur la même colonne ni dans le même carré *)
let no_interact (i,j,k) i1 j1 = (j1 <> j && square i1 j1 <> k)
(** Liste des couples (n,i) tels que n est sur la ligne i dans board *)
let no_conf board = Board.fold (fun (i,j) m l -> (m,i)::l) board []
(** met à jour la configuration, échoue si plus de solution *)
let update_configuration (i,j) n conf = let k = square i j in
  List.map
          (fun (n1,i1,s1) ->
                let s2 =
                  if n=n1 then IntSet.filter (no_interact (i,j,k) i1) s1
                  else if i=i1 then IntSet.filter (fun j1 -> j<>j1) s1
                  else s1
                in
                if IntSet.is_empty s2 then raise Error
                else (n1,i1,s2))
          conf
```

3. Construire une fonction `init_configuration` qui étant donné un jeu (objet de type `Board.t`) construit la configuration initiale.
   On pourra partir de `configuration0`, retirer tous les triplets (n, i, s) pour lesquels il existe dans le jeu une entrée n pour la position (i, j) et mettre à jour progressivement la configuration à l'aide de `update_configuration` en prenant en compte chaque entrée du jeu.
   Correction :

```
let init_configuration board =
  let g0 = configuration0
  in let nog = no_conf board
  in let g1 = List.filter (fun (n,i,_) -> not (List.mem (n,i) nog)) g0
  in let g = Board.fold update_configuration board g1
  in List.sort conf_compare g
```

4. Pour construire la solution on dispose d'un jeu b partiellement complété et d'une configuration compatible. Si dans la configuration, un des triplets a un ensemble vide de colonnes , alors le jeu n'a pas de solution. Sinon on peut choisir dans la configuration un triplet (n, i, s) et choisir une colonne j dans l'ensemble de colonnes s. On choisit alors de compléter le jeu avec l'entrée $(i, j) \mapsto n$, on met à jour le jeu et la configuration et on continue. Si ce nouveau jeu n'a pas de solution alors il faut revenir en arrière pour choisir une autre position pour n. Lorsqu'il n'y a plus de triplets dans la configuration c'est que le jeu est terminé.
   Pour programmer cette fonction on introduit deux exceptions :

```
exception Ok of Board.t
exception Error
```

   On lèvera l'exception (`Ok` b) si on a complété le jeu initial en b et on échouera avec l'exception `Error` si le jeu n'a pas de solution.
   Construire une fonction recursive `search` qui prend en argument un jeu b_0 et une configuration *conf* compatible et qui lève une exception `Ok` b avec b une solution pour compléter b_0 et renvoie `Error` si le jeu b_0 n'a pas de solution.
   L'algorithme sera le suivant :
   - si la configuration est vide alors le jeu est complet ;
   - sinon, s'il y a dans la configuration un triplet (n, i, s) avec s un ensemble vide alors le jeu n'a pas de solution ;
   - sinon on choisit (n, i, s) et $j \in s$ et on essaie le jeu dans lequel on pose n à la position (i, j). Si ce jeu n'a pas de solution alors on retire j de s et on continue.

   On pourra penser à trier la configuration par ordre croissant du nombre de colonnes associé à une entrée. On pourra utiliser :

```
val List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

   Le premier argument est une fonction `compare` telle que `compare` $x\ y = -1$ si $x < y$, `compare` $x\ y = 1$ si $x > y$ et 0 sinon.
   Correction :

```
let rec search board conf =
  match conf with
      [] -> raise (Ok board)
    |(n,i,cols)::og ->
       let j = IntSet.choose cols in
         try
           let og' = update_configuration (i,j) n og
```

```
      in let b = Board.add (i,j) n board
      in search b (List.sort conf_compare og')
    with Error ->
      let cols' = IntSet.remove j cols in
        if IntSet.is_empty cols' then raise Error
        else search board ((n,i,cols')::og)
```

5. En déduire une fonction `sudoku` qui étant donné un jeu initial, renvoie un jeu complété s'il existe et sinon échoue avec `Error`.

   Correction :

```
let sudoku board =
  try
  let g0 = init_configuration board
  in search board g0
  with Error -> (Printf.printf "Ce jeu n'a pas de solution\n"; raise Exit)
    | Ok b -> b
```

6. On a choisi une structure fonctionnelle (donc persistente) de map pour représenter le jeu et non pas un codage impératif par exemple par une matrice. Quel avantage a cette solution ?

   Correction : *Dans la fonction* `search`*, en cas d'erreur on reprend le calcul à partir d'un ancien état du jeu. Si celui-ci avait été modifié en place il serait nécessaire soit de faire des copies systématiques, soit de remettre à jour.*