Corrigé de l'examen de programmation fonctionnelle et logique

Mercredi 13 juin 2001, 14H00-17H00

Remarques et commentaires :
- Commencez par lire le sujet dans son intégralité.
- Les exercices sont indépendants les uns des autres. Les questions au sein d'un même exercice sont elles aussi indépendantes. Vous pouvez très bien ne pas savoir répondre à une question et réussir les suivantes.
- Écrivez lisiblement et en français, car les copies seront lues (anonymat oblige) !
- Facilitez la lecture et la compréhension des codes proposés.

Rappels

Les fonctionnelles sur liste prédéfinies (en `Ocaml`)

Les quatre fonctionnelles sur listes prédéfinies sont :
- `# List.iter;;`
  `- : ('a -> unit) -> 'a list -> unit = <fun>`
  `List.iter f [a1; ...; an]` est équivalent à `begin f a1; ...; f an; () end`.
- `# List.map;;`
  `- : ('a -> 'b) -> 'a list -> 'b list = <fun>`
  `List.map f [a1; ...; an]` est équivalent à `[f a1; ...; f an]`.
- `# List.fold_left;;`
  `- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>`
  `List.fold_left f a [b1; ...; bn]` est équivalent à `f (...(f (f a b1) b2) ...) bn`.
- `# List.fold_right;;`
  `- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>`
  `List.fold_right f [a1; ...; an] b` est équivalent à `f a1 (f a2 (... (f an b) ...))`.

Quelques fonctions de bases

- Concaténation de deux listes :
  `# [1;2]@[3;4];;`
  `- : int list = [1; 2; 3; 4]`
- Test d'égalité :
  `# let f a b = (a = b);;`
  `val f : 'a -> 'a -> bool = <fun>`
- `et` logique :
  `# true & false;;`
  `- : bool = false`
- `ou` logique :
  `# true or false;;`
  `- : bool = true`

Exercice 1 : typage d'expressions `Ocaml`

Donnez le type des expressions suivantes.
Important : la justification du résultat trouvé comptera autant que le résultat lui-même !

1.
```
# let rec f x y z = if x > 2.3 then y else z;;
# f;;
- : float -> 'a -> 'a -> 'a = <fun>
```
2.
```
# let rec f x y z = match x with
      t::r -> y t (f r y z)
    |   _  -> z;;
# f;;
- : 'a list -> ('a -> 'b -> 'b) -> 'b -> 'b = <fun>
```

3.
```
# exception Problème of float;;
```

```
# let rec a b c d =
   try
     match b with
       t::r -> if (c t d) then a r c t
                          else raise (Problème t)
     |  []  -> d
   with Problème f -> f;;
# a;;
- : float list -> (float -> float -> bool) -> float -> float = <fun>
```

Exercice 2 : manipulation de graphes `Ocaml`

On décide de représenter un graphe orienté par une liste de paires, chaque paire étant constituée d'un sommet (représenté par un entier) et de la liste de ses sommets successeurs (eux-mêmes représentés par des entiers). La figure 1 présente un exemple de graphe dont voici la transcription en `Ocaml` suivant le codage proposé :

```
# let g = [(1,[2;3]);(2,[4]);(3,[4]);(4,[1;5]);(5,[])];;
val g : (int * int list) list =
[(1, [2; 3]); (2, [4]); (3, [4]); (4, [1; 5]); (5, [])]
```

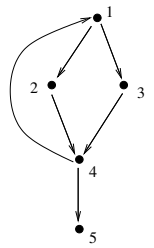

FIG. 1 – Exemple de graphe.

Quand l'énoncé laisse la liberté d'utiliser ou non des fonctionnelles ou des exceptions, toute solution correcte utilisant ces mécanismes se verra gratifiée d'un bonus.

1. **Petites fonctions auxiliaires.**

   (a) Écrivez une fonction `tête` qui prend en entrée une paire composée d'un sommet et de la liste des successeurs de ce sommet, et qui renvoie le sommet.

   ```
   # let tête x = match x with (a,_) -> a;;
   val tête : 'a * 'b -> 'a = <fun>
   ```

   Exemple d'utilisation :

   ```
   # tête (4,[1;5]);;
   - : int = 4
   ```

   (b) Écrivez une fonction `successeurs` qui prend en entrée une paire composée d'un sommet et de la liste des successeurs de ce sommet, et qui renvoie la liste des successeurs du sommet.

   ```
   # let successeurs x = match x with (_,a) -> a;;
   val successeurs : 'a * 'b -> 'b = <fun>
   ```

   Exemple d'utilisation :

   ```
   # successeurs (4,[1;5]);;
   - : int list = [1; 5]
   ```

   (c) Écrivez une fonction `appartient` à valeurs booléennes qui prend en entrée un objet et une liste et qui renvoie la valeur vraie si et seulement si l'objet appartient à la liste.

```
# let rec appartient x l = match l with
    t::r -> (x=t) or (appartient x r)
  | _   -> false;;
val appartient : 'a -> 'a list -> bool = <fun>
```

```
# let appartient x l =
    List.fold_left (function y -> function z -> y or (x=z)) false l;;
val appartient : 'a -> 'a list -> bool = <fun>
```

```
# exception Found;;
exception Found
```

```
# let appartient x l =
  try
   List.fold_left
     (function y -> function z -> if (x=z) then raise Found else y)
     false
     l
  with Found -> true;;
val appartient : 'a -> 'a list -> bool = <fun>
```

(d) Écrivez une fonction `simplifie` qui prend en entrée une liste `l` et qui renvoie une liste qui contient exactement une et une seule fois chacun des éléments présents dans `l`. (L'ordre dans lequel les éléments apparaissent dans la solution n'a strictement aucune importance.)

```
# let rec simplifie = function
    []  -> []
 | t::r -> if (appartient t r) then simplifie r
                               else t::(simplifie r);;
val simplifie : 'a list -> 'a list = <fun>
```

```
# simplifie [1;5;3;5;4;6;7;8;4];;
- : int list = [1; 3; 5; 6; 7; 8; 4]
```

```
# let simplifie l =
   List.fold_left
     (function x -> function y -> if (appartient y x) then x else y::x)
     []
     l;;
val simplifie : 'a list -> 'a list = <fun>
```

```
# simplifie [1;5;3;5;4;6;7;8;4];;
- : int list = [8; 7; 6; 4; 3; 5; 1]
```

```
# let simplifie l =
   let rec aux input output = match input with
     t::r -> if (appartient t output)
             then aux r output
             else aux r (output@[t])
    | _  -> output
   in aux l [];;
val simplifie : 'a list -> 'a list = <fun>
```

```
# simplifie [1;5;3;5;4;6;7;8;4];;
- : int list = [1; 5; 3; 4; 6; 7; 8]
```

Exemple d'utilisation :

```
# simplifie [1;5;3;5;4;6;7;8;4];;
- : int list = [1; 5; 3; 4; 6; 7; 8]
```

2. **Ensembles de sommets**

(a) Nous voulons une fonction `touteslestêtes` qui prend en entrée un graphe, et qui renvoie la liste de toutes les têtes des paires inclues dans la définition du graphe.

i. Écrivez une version récursive de cette fonction.

```
# let rec touteslestêtes x = match x with
    t::r -> (tête t)::(touteslestêtes r)
  |   _  -> [];;
val touteslestêtes : ('a * 'b) list -> 'a list = <fun>

# touteslestêtes g;;
- : int list = [1; 2; 3; 4; 5]
```

ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let touteslestêtes x = List.map tête x;;
val touteslestêtes : ('a * 'b) list -> 'a list = <fun>

# touteslestêtes g;;
- : int list = [1; 2; 3; 4; 5]
```

Exemple d'utilisation :

```
# touteslestêtes g;;
- : int list = [1; 2; 3; 4; 5]
```

(b) Écrivez une fonction `touslessuccesseurs` qui prend en entrée un graphe, et qui renvoie la liste de tous les sommets qui apparaissent dans au moins une liste de successeurs.

```
# let rec touslessuccesseurs x = match x with
    t::r -> (successeurs t)@(touslessuccesseurs r)
  |   _  -> [];;
val touslessuccesseurs : ('a * 'b list) list -> 'b list = <fun>

# touslessuccesseurs g;;
- : int list = [2; 3; 4; 4; 1; 5]

# let touslessuccesseurs x =
   List.fold_right (function a-> function b-> (successeurs a)@b ) x [];;
val touslessuccesseurs : ('a * 'b list) list -> 'b list = <fun>

# touslessuccesseurs g;;
- : int list = [2; 3; 4; 4; 1; 5]
```

Exemple d'utilisation :

```
# touslessuccesseurs g;;
- : int list = [2; 3; 4; 4; 1; 5]
```

3. **Bonne définition des graphes**

(a) Pour que la définition d'un graphe soit cohérente, il faut que tous les sommets qui apparaissent dans sa définition soient la tête d'une paire. Autrement dit, tout sommet qui appartient à une liste de successeurs doit aussi être la tête d'une paire. En vous aidant des fonctions précédentes, écrivez une fonction `cohérent` à valeurs booléennes qui est vraie si et seulement si tous les sommets qui apparaissent dans sa définition sont la tête d'une paire.

i. Écrivez une version récursive de cette fonction.

```
# let cohérent x =
   let t = touteslestêtes x and s = touslessuccesseurs x in
     let rec check l = match l with
         a::b -> (appartient a t) & (check b)
       |  _   -> true
     in check s;;
val cohérent : ('a * 'a list) list -> bool = <fun>
```

ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let cohérent x =
   List.fold_right
     (function a -> function b-> b & (appartient a (touteslestêtes x)))
     (touslessuccesseurs x)
```

```
      true;;
val cohérent : ('a * 'a list) list -> bool = <fun>

# cohérent g;;
- : bool = true

# cohérent [(1,[2])];;
- : bool = false

# let cohérent x =
  let têtes = touteslestêtes x in
   List.fold_right
     (function a -> function b-> b & (appartient a (têtes)))
     (touslessuccesseurs x)
     true;;
val cohérent : ('a * 'a list) list -> bool = <fun>

# cohérent g;;
- : bool = true

# cohérent [(1,[2])];;
- : bool = false
```

Exemples d'utilisation :

```
# cohérent g;;
- : bool = true

# cohérent [(1,[2])];;
- : bool = false
```

(b) En utilisant le mécanisme des exceptions, nous voulons optimiser la fonction `cohérent` pour qu'elle arrête de s'exécuter dès qu'un élément fautif est trouvé (sommet qui appartient à une liste des successeurs, mais qui n'est pas la tête d'une paire).

i. Optimisez la version récursive de la fonction `cohérent`.

```
# exception Fautif;;
exception Fautif

# let cohérent_optimisé x =
   let t = touteslestêtes x and s = touslessuccesseurs x in
     let rec check l = match l with
         a::b -> if (appartient a t) then (check b)
                                     else raise Fautif
       |  _   -> true
   in try (check s) with Fautif -> false;;
val cohérent_optimisé : ('a * 'a list) list -> bool = <fun>

# cohérent_optimisé g;;
- : bool = true

# cohérent_optimisé [(1,[2])];;
- : bool = false
```

ii. Optimisez la version de la fonction `cohérent` qui utilise une des fonctionnelles prédéfinies.

```
# exception Fautif;;
exception Fautif

# let cohérent_optimisé x =
   try
   List.fold_right
     (function a -> function b-> if (appartient a (touteslestêtes x))
                                 then b
                                 else raise Fautif)
     (touslessuccesseurs x)
```

```
      true
    with Fautif -> false;;
val cohérent_optimisé : ('a * 'a list) list -> bool = <fun>

# cohérent_optimisé g;;
- : bool = true

# cohérent_optimisé [(1,[2])];;
- : bool = false
```

(c) Nous voulons maintenant une fonction `fautif`, qui prend en entrée un graphe et qui renvoie la valeur d'un élément fautif (sommet qui appartient à une liste des successeurs, mais qui n'est pas la tête d'une paire). Tout comme la fonction `cohérent_optimisé`, cette fonction doit *obligatoirement* être optimisée au moyen d'exceptions. Si le graphe ne contient pas d'élément fautif, la fonction devra renvoyer un message d'erreur.

i. Version récursive de cette fonction.

```
# exception Fautif of int;;
exception Fautif of int

# let fautif x =
    let t = touteslestêtes x and s = touslessuccesseurs x in
      let rec check l = match l with
          a::b -> if (appartient a t) then (check b)
                                      else raise (Fautif a)
        |  _   -> failwith "Pas d'élément fautif dans ce graphe"
    in try (check s) with Fautif c -> c;;
val fautif : (int * int list) list -> int = <fun>
```

ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# exception Fautif of int;;
exception Fautif of int

# let fautif x =
    try
    List.iter
      (function a -> if (appartient a (touteslestêtes x))
                        then ()
                        else raise (Fautif a))
      (touslessuccesseurs x);
      failwith "Pas d'élément fautif dans ce graphe"
    with Fautif c -> c;;
val fautif : (int * int list) list -> int = <fun>

# fautif g;;
Exception: Failure "Pas d'élément fautif dans ce graphe".

# fautif [(1,[2])];;
- : int = 2
```

Exemples d'utilisation :

```
# fautif g;;
Exception: Failure "Pas d'élément fautif dans ce graphe".

# fautif [(1,[2])];;
- : int = 2
```

(d) Écrivez une fonction *différentes* à valeurs booléennes qui prend en entrée deux listes `l1` et `l2` et qui renvoie `true` si et seulement si au moins un des éléments de `l1` n'appartient pas à `l2`. Cette fonction devra `obligatoirement` utiliser une des fonctionnelles prédéfinies (et non la récursivité), ainsi qu'une exception pour optimiser l'emploi de la fonctionnelle.

```
# exception Trouvé;;
exception Trouvé
# let différentes l1 l2 =
  try
   List.fold_left
    (function x -> function y -> if (not (appartient y l2)) then raise Trouvé
                                                             else x)
    false
    l1
  with Trouvé -> true;;
val différentes : 'a list -> 'a list -> bool = <fun>
```

Exemples d'utilisation :

```
# différentes [1;2;3] [2];;
- : bool = true
# différentes [1;2;3;2] [1;2;3];;
- : bool = false
```

4. **Ensemble des sommets atteignables à partir d'un sommet donné**

   (a) Nous voulons une fonction `suivants` qui prenne en entrée un graphe et un sommet et qui renvoie la liste des successeurs (directs) du sommet dans le graphe.
   Écrivez une version récursive de cette fonction.

```
# let rec suivants graph s = match graph with
    (a,b)::r -> if (a=s) then b else (suivants r s)
  |      _   -> failwith "Ce sommet ne figure pas dans le graphe";;
val suivants : ('a * 'b) list -> 'a -> 'b = <fun>
```

   Exemple d'utilisation :

```
# suivants g 2;;
- : int list = [4]
```

   (b) Écrivez une fonction prenant en entrée un graphe et une liste `l` de sommets, et qui renvoie une liste contenant tous les sommets de `l` et tous les successeurs des sommets de `l`. Vous pourrez utiliser la fonction `simplifie` (question 1d) pour nettoyer le résultat. Votre fonction devra utiliser une des fonctionnelles prédéfinies.

```
# let extension g l =
  simplifie
   (List.fold_right (function x -> function y -> y @ (suivants g x)) l l);;
val extension : ('a * 'a list) list -> 'a list -> 'a list = <fun>
```

   Exemple d'utilisation :

```
# extension g [1;2];;
- : int list = [1; 2; 4; 3]
```

   (c) Nous voulons maintenant une fonction récursive `touslessuivants` qui prenne en entrée un graphe et un sommet `s` et qui renvoie la liste de tous les sommets qui peuvent être atteints à partir de `s`, directement ou transitivement (voir les exemples d'utilisation). *Attention* à ne pas écrire une fonction qui boucle indéfiniment.

```
# let touslessuivants g s =
   let rec allongement l =
     let newl = extension g l
     in if (différentes newl l) then (allongement newl)
                                else l
   in allongement [s];;
val touslessuivants : ('a * 'a list) list -> 'a -> 'a list = <fun>
```

   Exemples d'utilisation :

```
# touslessuivants g 2;;
- : int list = [2; 4; 1; 5; 3]
# touslessuivants g 5;;
- : int list = [5]
```

Manipulation de graphes en SWI-Prolog

On décide de représenter un graphe orienté par une liste de sous-listes, chaque sous-liste étant constituée d'un sommet (représenté par un entier) et de la liste de ses sommets successeurs (eux-mêmes représentés par des entiers). La figure 1 présente un exemple de graphe dont voici la transcription en `SWI-Prolog` suivant le codage proposé :

```
[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]],[5,[]]]
```

1. **Petites fonctions auxiliaires.**

   (a) Écrivez un prédicat `appartient` qui prend en entrée un objet et une liste et qui est vrai si et seulement si l'objet appartient à la liste.

   ```
   appartient(X,[X|_]).
   appartient(X,[_|T]):-appartient(X,T).
   ```

   (b) Écrivez un prédicat `appartientopt`, nouvelle version de `appartient` optimisée au moyen de coupures.

   ```
   appartientopt(X,[X|_]):-!.
   appartientopt(X,[_|T]):-appartientopt(X,T).
   ```

   (c) Écrivez un prédicat `nappartientpas` qui prend en entrée un objet et une liste et qui est vrai si et seulement si l'objet *n'*appartient *pas* à la liste. Cette fonction doit *obligatoirement* utiliser le mécanisme de coupure (vous n'utiliserez pas ici la fonction « différent » : « =\= », ni le prédicat `not`).

   ```
   nappartientpas(_,[]):-!.
   nappartientpas(X,[X|_]):-!,fail.
   nappartientpas(X,[_|T]):-nappartientpas(X,T).
   ```

   (d) Écrivez un prédicat `sousliste` qui prend en entrée deux listes `A` et `B` et qui est vrai si et seulement si tous les éléments de `A` sont éléments de `B`.

   ```
   sousliste([],_).
   sousliste([T|R],L):-appartient(T,L), sousliste(R,L).
   ```

   Exemples d'utilisation :

   ```
   ?- sousliste([2,3,4,4,1,5], [1,2,3,4,5]).
   Yes
   ?- sousliste([2,3,4,4,1,5], [1,2,4,5]).
   No
   ```

   (e) Écrivez un prédicat `successeur` qui prend en entrée deux sommets `A` et `B` et un graphe `G` et qui est vrai si et seulement si `B` est un successeur direct de `A` dans `G`.

   ```
   successeur(A,B,[[A,L]|_]):-appartient(B,L).
   successeur(A,B,[_|L]):-successeur(A,B,L).
   ```

   Exemples d'utilisation :

   ```
   ?- successeur(1,3,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).
   Yes
   ?- successeur(1,4,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).
   No
   ```

   (f) Votre prédicat `successeur` est-il capable de générer les successeurs d'un sommet donné comme dans l'exemple ci-dessous ?

   ```
   ?- successeur(1,X,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).
   X = 2 ;
   X = 3 ;
   No
   ```

   Pourquoi ? (Répondre en trois lignes au grand maximum.)

2. **Ensemble de sommets**
   Écrivez un prédicat `touteslestetes` qui prend en entrée une liste et un graphe et qui est vrai si et seulement si la liste est la liste des sommets du graphe.

```
touteslestetes([],[]).
touteslestetes([X|T],[[X|_]|R]):-touteslestetes(T,R).
```

Exemples d'utilisation :

```
?- touteslestetes(X,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]],[5,[]]]).
X = [1, 2, 3, 4, 5]
Yes
?- touteslestetes([2,3,4,5], [[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]],[5,[]]]).
No
```

3. **Bonne définition des graphes**

   (a) Écrivez un prédicat `coherent` qui prend en entrée un graphe et qui est vrai si et seulement si le graphe est bien défini, c'est-à-dire si et seulement si tous les sommets qui apparaissent dans une liste de sommets successeurs apparaissent également comme sommets.

   Nous supposerons ici que nous avons à notre disposition un prédicat `touslessuccesseurs` qui prend en entrée une liste et un graphe et qui est vrai si et seulement si la liste est la concaténation des listes de successeurs du graphe.

```
coherent(G):-touteslestetes(T,G),touslessuccesseurs(S,G),sousliste(S,T).
```

   Exemples d'utilisation :

```
?- coherent([[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]],[5,[]]]).
Yes
?- coherent([[1,[2,3]],[2,[4]]]).
No
```

   (b) Optimisez le prédicat `coherent` au moyen de coupures.

```
coherent(G):-touteslestetes(T,G),touslessuccesseurs(S,G),!,sousliste(S,T),!.
```

4. **Existence d'un chemin reliant deux sommets**

   (a) Nous voulons un prédicat `chemin` qui prend en entrée deux sommets `A` et `B` et un graphe `G` et qui est vrai si et seulement si le graphe `G` contient un chemin de `A` vers `B`. *Indication* : pour résoudre cette question, vous pourrez supposer que votre prédicat `successeur` est capable de générer les successeurs d'un sommet.

```
chemin(A,A,_).
chemin(A,B,G):-successeur(A,B,G).
chemin(A,B,G):-successeur(A,C,G),chemin(C,B,G).
```

   Exemples d'utilisation :

```
?- chemin(1,4,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).
Yes
?- chemin(5,4,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).
No
```

   (b) Votre prédicat `chemin` est-il interrogeable par la question suivante ?

```
?- chemin(1,X,[[1,[2,3]],[2,[4]],[3,[4]],[4,[1,5]]]).

X = 1 ;
X = 2 ;
X = 3 ;
X = 2 ;
X = 4 ;
X = 4 ;
etc.
```

   (c) Écrivez un prédicat `antichemin` qui prend en entrée deux sommets `A` et `B` et un graphe `G`, et qui est vrai si et seulement si il n'existe pas dans le graphe `G` de chemin de `A` vers `B`.

```
antichemin(A,B,G):-chemin(A,B,G),!,fail.
antichemin(_,_,_).
```