Examen de programmation fonctionnelle et logique

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 f g x = g(g x);;`

   ```
   f : ('a -> 'a) -> 'a -> 'a
   ```

2. ```
   let rec f a b c =
     match a with
         []   -> [b c]
      | t::r  -> (b t)::(f r b c);;
   ```

   ```
   f : 'a list -> ('a -> 'b) -> 'a -> 'b list
   ```

3. ```
   exception Problème;;
   let rec f g x y =
    try
      match y with
            []   -> if (g x) > x  then (g x)
                                  else raise Problème
         | t::r  -> f g (g t) r
    with Problème -> f g (x-1) y;;
   f : (int -> int) -> int -> int list -> int
   ```

Exercice 2: les itérateurs sur liste en `ocaml`

Quatre itérateurs sur listes sont définis : `List.iter`, `List.map`, `List.fold_left` et `List.fold_right`. Les types de ces fonctionnelles et leurs comportements sont les suivants :

`List.iter : ('a -> 'b) -> 'a list -> unit`.
    `List.iter f [a1; ...; an]` est équivalent à `begin f a1; ...; f an; () end`.

`List.map : ('a -> 'b) -> 'a list -> 'b list`.
    `List.map f [a1; ...; an]` est équivalent à `[f a1; ...; f an]`.

`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.
    `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` .
    `List.fold_right f [a1; ...; an] b` est équivalent à `f a1 (f a2 (... (f an b) ...))`.

Écrivez une fonction qui réalise `List.iter`, une qui réalise `List.map`, une qui réalise `List.fold_left` et une qui réalise `List.fold_right`. Les quatre fonctions demandées doivent bien évidemment être écrites en `ocaml`.

```
let rec iter f = function
    []  -> ()
 | t::r -> begin f t; iter f r end;;
```

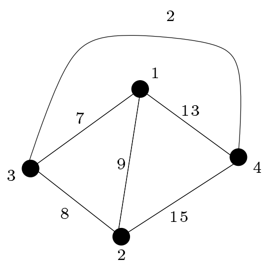

FIG. 1: Exemple de problème avec quatre villes.

```
let rec map f = function
    []  -> []
 | t::r -> (f t)::(map f r);;
```

```
let rec fold_left f a = function
    []  -> a
 | t::r -> fold_left f (f a t) r;;
```

```
let rec fold_right f l b = match l with
    []  -> b
 | t::r -> f t (fold_right f r b);;
```

Exercice 3: le voyageur de commerce (en `ocaml`)

Un voyageur de commerce doit visiter N villes. Il peut les parcourir dans l'ordre de son choix, du moment qu'il visite chaque ville exactement une fois et qu'il termine son parcours dans la ville de départ. Son problème est de minimiser la distance totale parcourue.

Les données du problème sont ici :

- Le nombre de villes, n. Les villes sont numérotées de 1 à n. ($n = 4$ dans l'exemple de la figure 1.)
- Une liste de distances entre les villes, notée ld. Chaque distance est décrite au moyen du type `chemin` :

  ```
  type chemin = Chemin of (int * int list);;
  ```

  qui contient une paire constituée d'un entier (la longueur du chemin) et d'un liste d'entiers (la liste des villes visitées par le chemin, ordonnées dans l'ordre de visite). Exemple d'une valeur possible de ld dans l'exemple de la figure 1 : $[Chemin(9, [1; 2]); Chemin(8, [2; 3]); Chemin(7, [1; 3]); Chemin(13, [4; 1]); Chemin(15, [4; 2]); Chemin(2, [4; 3])]$.

Conventions :

- Pour faciliter l'écriture des programmes, il est décidé que la liste d'entiers d'un objet de type `chemin` représente la liste des villes visitées, de la ville visitée en dernier (tête de liste) à la ville de départ (queue de la liste). Ainsi le chemin `Chemin(24,[1;4;2])` part de la ville 2, puis traverse la ville 4 avant d'aboutir à la ville 1.
- Pour alléger l'énoncé on appellera **chemin élémentaire** un objet de type `chemin` dont la liste d'entiers contient exactement deux entiers. Typiquement, ld est une liste de chemins élémentaires.

1. Écrivez une fonction `longueur` qui calcule la longueur d'une liste.

   ```
   let rec longueur = function
       []  -> 0
    | _::r -> 1 + (longueur r);;
   ```

2. Écrivez une fonction `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.

   ```
   let rec nappartientpas x = function
       []  -> true
    | t::r -> (t <> x) & (nappartientpas x r);;
   ```

3. Écrivez une fonction `debut` qui prend en entrée une liste de chemins élémentaires et qui retourne la liste de tous les chemins élémentaires possibles et commençant par la ville numéro 1.
   Exemple : appelée sur la liste *ld* proposée ci-dessus, la fonction `debut` doit renvoyer la liste : `[Chemin(9,[2;1]) ; Chemin(7,[3;1]) ; Chemin(13,[4;1])]`.

```
let rec debut = function
    []  -> []
 | t::r -> let (Chemin(long,[x;y])) = t in
            let but = if x=1 then y else x in
              if ((x=1) or (y=1))
                 then (Chemin(long,[but;1]))::(debut r)
                 else (debut r);;
```

4. Écrivez une fonction `prolonge` qui prend en entrée un chemin (noté c) et un chemin élémentaire et qui prolonge c d'une ville, au moyen du chemin élémentaire, si c'est possible, c'est-à-dire si la tête de c est une des extrémités du chemin élémentaire et si les deux extrémités du chemin élémentaire ne sont pas déjà incluses dans c. La fonction `prolonge` réalise donc une sorte de concaténation de chemins.

```
let prolonge c ce =
 let (Chemin(l,t::r)) = c and (Chemin(le,[x;y])) = ce
  in let but = if x=t then y else x
   in if ((x=t) or (y=t)) & (nappartientpas but (t::r))
        then (Chemin(l+le,but::t::r))
        else failwith "prolonge";;
```

5. Écrivez une fonction `étend` qui prend en entrée un chemin (noté c) et une liste de chemins élémentaires (notée l) et qui construit la liste des chemins obtenus en prolongeant c d'une ville au moyen d'un chemin élémentaire de l. `étend` utilisera bien évidemment `prolonge`. Elle devra aussi mettre en œuvre un itérateur sur liste.

```
let étend c l =
 let prolongateur chem liste =
       try (prolonge c chem)::liste with Failure _ -> liste
  in List.fold_right prolongateur l [];;
```

6. Écrivez une fonction `extension` qui prend en entrée une liste l de chemins et une liste le de chemins élémentaires et qui produit la liste de tous les chemins obtenus par prolongement d'un chemin de l au moyen d'un chemin de le.

```
let extension l le =
 let extenseur chem liste = (étend chem le)@liste
  in List.fold_right extenseur l [];;
```

7. Écrivez une fonction `tousleschemins` qui prend en entrée n et ld et qui construit tous les chemins possibles contenant exactement n villes et commençant par la ville numéro 1.

```
let tousleschemins n ld =
 let rec tousaux = function
       []  -> tousaux (debut ld)
    | ((Chemin(_,t))::r as l) -> if (longueur t) >= n
                                 then l
                                 else tousaux (extension l ld)
 in if n <= 1
       then failwith "tousleschemins"
       else tousaux [];;
```

8. Maintenant que nous avons tous les chemins commençant par la ville numéro 1, nous voulons avoir tous les circuits possibles. Vous devez donc écrire une fonction `touslescircuits` construisant tous ces circuits.
   Attention : un détail interdit d'utiliser (directement ou indirectement) la fonction `prolonge` précédemment définie. Vous n'êtes pas obligés d'écrire des versions modifiées de fonctions existantes dont vous pourriez avoir besoin, vous pouvez vous contentez d'indiquer les différences entre les différentes versions.

```
let prolongesoft c ce =
 let (Chemin(l,t::r)) = c and (Chemin(le,[x;y])) = ce
  in let but = if x=t then y else x
   in if ((x=t) or (y=t)) & (but=1)
         then (Chemin(l+le,but::t::r))
         else failwith "prolongesoft";;

let étendsoft c l =
 let prolongateur chem liste =
       try (prolongesoft c chem)::liste with Failure _ -> liste
  in List.fold_right prolongateur l [];;

let extensionsoft l le =
 let extenseur chem liste = (étendsoft chem le)@liste
  in List.fold_right extenseur l [];;

let touslescircuits n ld =
    extensionsoft (touslescheminsn n ld) ld;;
```

9. Écrivez une fonction `plus_court` qui prend en entrée une liste de chemins et qui renvoie celui de plus courte distance.

```
let plus_court l =
 let rec plus_court_aux pc = function
        []  -> pc
     | (Chemin(l,_) as chem)::r -> let (Chemin(lpc,_))=pc
                                   in if l < lpc
                                         then plus_court_aux chem r
                                         else plus_court_aux  pc  r
 in if l = [] then failwith "plus_court"
             else let tl::rl = l in plus_court_aux tl rl;;
```

10. Écrire une fonction qui résout le problème !

```
let plus_court_circuit n ld =  plus_court(touslescircuits n ld);;
```

Exercice 4: le voyageur de commerce (en SWI-Prolog)

Un voyageur de commerce doit visiter N villes. Il peut les parcourir dans l'ordre de son choix, du moment qu'il visite chaque ville exactement une fois et qu'il termine son parcours dans la ville de départ. Son problème est de minimiser la distance totale parcourue.

Les données du problème sont ici :

- Le nombre de villes, N. ($N = 4$ dans l'exemple de la figure 1.)
- Une liste de distances entre les villes, notée LD. Chaque distance est décrite par une liste de trois entiers : le numéro de la ville de départ, celui de la ville d'arrivée et la distance séparant ces deux villes. Les villes sont numérotées de 1 à N. ($LD = [[1, 2, 9], [2, 3, 8], [1, 3, 7], [4, 1, 13], [4, 2, 15], [4, 3, 2]]$ dans l'exemple de la figure 1.)

1. Écrivez un prédicat `longueur_liste` qui calcule la longueur d'une liste.

```
longueur_liste([],0).
longueur_liste([_|R],L) :- longueur_liste(R,M), L is M+1.
```

2. Écrivez un prédicat `tous_elements` qui prend en entrée le nombre N de villes et qui vérifie que la liste ne contient que des entiers représentant des numéros de villes.

```
tous_elements([],_).
tous_elements([T|R],N) :- between(1,N,T), tous_elements(R,N).
```

3. Écrivez un prédicat `tous_differents` qui vérifie que les éléments d'une liste sont des nombres deux à deux distincts.

```
tous_differents([]).
tous_differents([T|R]):- pas_element(T,R), tous_differents(R).

pas_element(_,[]).
pas_element(X,[T|R]):- X =\= T, pas_element(X,R).
```

4. Écrivez un prédicat `dernier` qui permet d'extraire le dernier élément d'une liste.

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

5. Écrivez un prédicat `circuit` qui construit une liste de villes représentant le chemin que doit parcourir le voyageur de commerce (sans tenir compte pour l'instant de la minimisation).

```
circuit([T|R],N) :- longueur_liste(R,N), !, tous_elements(R,N),
                    tous_differents(R), dernier(R,T).
```

6. Écrivez un prédicat `distance` qui lit, dans la liste des distances LD, la distance entre deux villes données.

```
distance(X,Y,[[X,Y,D]|_],D) :- !.
distance(X,Y,[[Y,X,D]|_],D) :- !.
distance(X,Y,[[_,_,_]|R],L) :- distance(X,Y,R,L).
```

7. Écrivez un prédicat `longueur_circuit` qui calcule la longueur d'un circuit au moyen de la table des distances.

```
longueur_circuit([_],_,0).
longueur_circuit([X|[Y|R]],T,L) :- distance(X,Y,T,J),
                                   longueur_circuit([Y|R],T,K), L is J + K.
```

8. Écrivez un prédicat `existe_plus_petit` qui prend en entrée un circuit et qui est vrai si il existe un circuit plus court satisfaisant les hypothèses.

```
existe_plus_petit(C,T,N) :- longueur_circuit(C,T,L),
                            circuit(P,N), longueur_circuit(P,T,S),
                            S < L.
```

9. Écrivez un prédicat `plus_court_circuit` qui construit toutes les solutions au problème du voyageur de commerce.

```
plus_court_circuit(L,S,T,N) :- circuit(L,N), plus_court(L,T,N),
                               longueur_circuit(L,T,S).
plus_court(L,T,N) :- existe_plus_petit(L,T,N), !, fail.
plus_court(_,_,_).
```

10. Question subsidiaire : donnez au moins un exemple d'optimisation d'un des prédicats précédents au moyen de coupure.