Algorithmique-Programmation : IAP1 - feuille d'exercices numéro 1 : corrigé

Les exercices proposés sont volontairement nombreux, de manière à ce que chacun puisse y trouver son compte. Les prendre dans l'ordre. Les exercices ou questions marqués d'une étoile sont un peu plus difficile et peuvent être passés dans un premier temps. L'étudiant y reviendra ensuite.
Pour chaque question, il est demandé d'écrire, avant toute chose, une interface (en suivant le modèle donné en cours). Vous vérifierez ce type avec la réponse de Ocaml : si il y a des différences réfléchissez un peu .....
Chaque fonction sera testée : cas nominaux, cas erronnés. Mettre les tests et leurs résultats en commentaire dans votre fichier OCaml. L'exemple de la fonction valeur absolue est donnée ci-dessous. Dans ce cas, on peut explorer les cas de tests suivants : un entier négatif (-5 par exemple), 0 et un entier positif (3 par exemple). En effet la spécification d'une telle fonction (bien connue des mathématiciens) montre bien que le résultat dépend du signe de l'entier. D'autre part, 0 est un cas de test bien légitime car est-il positif ? négatif ? une erreur dans la manipulation des signes de comparaison est si vite arrivée ....

```
(* Interface abs :
type int -> int
arg a
post abs a = valeur absolue de a
tests  abs(-5), abs 0, abs 3
*)
let abs a = if a <0 then (-a) else a;;
```

Exercice 1 (**Fonctions simples**)

1. Ecrire une fonction qui teste si son argument est pair. On indique que l'expression *a mod b* retourne le reste dans la division entière de *a* par *b*.
   Remarque : **qui teste** signifie la plupart du temps **qui retourne true si la condition est vraie et false sinon**.

   ```
   (*Interface est_pair
   type : int -> bool
   arguments : x
   pré-condition : rien
   post-condition : le résultat est true si x est pair, false sinon
   tests : est_pair(5); est_pair(52);est_pair(-6);est_pair(-5)*)
   let  est_pair x = x mod 2 = 0;;
   ```

   Ici pas de probleme particulier sauf peut-être l'utilisation des 2 symboles = qui n'ont pas la même signification. Le premier est une liaison d'un nom et d'une valeur (celle de l'expression `x mod 2 = 0`*) et le second est l'opérateur de comparaison.*

   L'écriture suivante est à rejeter :

   ```
   let  est_pair x = if x mod 2 = 0 then true else false;;
   ```

2. Ecrire une fonction qui retourne -1 si son argument est négatif, 0 si c'est 0 et 1 si l'argument est positif.

   ```
   (* Interface signe :
   type : int -> int
   arguments : x
   ```

```
pre : rien
post : signe x =  -1 si x est négatif, 0 si x=0 et 1 si x est positif.
Tests :  signe 0 ; signe (-6) ; signe 4
*)

let signe x = if x < 0 then (-1)
              else  if x=0 then 0 else  1 ;;
```

3. Ecrire une fonction qui calcule le volume d'une sphère

   On peut envisager plusieurs façons de définir la fonction selon que l'on déclare la constante pi localement ou globalement. En tout cas on insiste sur l'intéret de nommer cette valeur. Ci dessous 3 solutions : préférer les 2 dernières qui encapsulent la constante pi qui a priori n'est nécessaire que pour calculer le volume de la sphère. Remarquons que dans la 3ème écriture, cube est une fonction locale.

```
let pi = 3.14;;

(* Interface sphere
type float -> float
argument : r le rayon de la spère
precondition : r positif ou nul
postcondition : (sphere r)= volume de la sphere de rayon r
Tests : sphere 3.4
*)
let sphere r =  4. /. 3. *. pi *. r *. r *. r;;

let sphere r = let pi = 3.14 in
                  4. /. 3. *. pi *. r *. r *. r;;

let sphere r = let pi = 3.14 in
                  let cube x = x *. x *. x in
                  4. /. 3. *. pi *. (cube r);;
```

4. Ecrire une fonction qui prend en argument 3 entiers et retourne le plus grand de ces entiers.

```
*(Interface max3 :
type 'a *'a *'a -> 'a
arguments : (a,b,c)
postcondition : max(a,b,c) = le plus grand des 3 entiers
Tests : (*place 3*)
         max3 (3,4,5); max3 (4,3,5);
        (*place 2*)
         max3 (2,5,4); max3 (4,5,3);
        (*place 1*)
         max3 (5,4,2); max3 (5, 2,4);
        (* 2 égaux *)
        max3 (3,3,1)
        (3 égaux)
        max3 (-6, -6, -6)
*)
let max3 (a,b,c) = if a>= b then if a >= c then a else c
                            else if b >= c then b else c;;
```

Autre solution : Proposer la solution qui consiste à écrire une fonction max2 (attention la fonction max prédéfinie est currifyée) qui calcule le plus grand de 2 nombres puis l'utiliser pour écrire max3

```
let max2 (a,b) = if a >= b then a else b;;

let max3 (a,b,c) = max2  (a, max2 (b,c));;
```

Pour les tests, envisagez les différents ordres possibles pour les nombres et compléter pour être exhaustif avc 3 différents, 2 différents et tous égaux.

Dernière chose : remarquons le type de la fonction : `('a * 'a * 'a* -> 'a`. *L'opérateur de comparaison est polymorphe : on peut appliquer cet opérateur sur tous les types.*

5. Ecrire une fonction qui ajoute de part et d'autre d'une chaîne de caractères quelconque la chaîne "!!" appelée cadre.

   Modifier la fonction de manière à ce que le cadre devienne aussi un argument de la fonction (on dit que l'on *abstrait* le cadre).

   On veut maintenant encadrer dissymétriquement. Introduire les paramètres nécessaires et écrire la nouvelle fonction.

```
let encadrer s = let cadre = "!!" in cadre ^ s ^ cadre;;

let encadrer2 (s,cadre) =  cadre ^ s ^ cadre;;

let encadre3 (gauche, s, droit) =  gauche ^ s ^ droit;;
```

6. Ecrire une fonction qui détermine le nombre de solutions réelles d'une équation du second degré (0 quand pas de solution réelle, 1 si racine double, 2 sinon).

   Une équation de la forme $ax^2 + bx + c = 0$ est parfaitement définie par la donnée des coefficients a, b et c. Faie remarquer que le coeff a doit être non nul sinon le problème n'a pas de sens (équation du 1er degré). Que faire dans ce cas ? la spec ne dit rien. Par exemple retourner -1 dans un tel cas ou mieux faire échouer la fonction. Elle n'est pas définie dans ce cas.

```
(*Interface solutions
type : float*float*float  -> int
arguments : (a,b,c) coefficients d'une équation du second degré
pre : a<>0 (sinon ce n'est pas du second degré)
poscondition : solutions(a,b,c)= nombre de solutions r\'eelles
raises : Failure "solutions : premier degre" si a=0
Tests : (*envisager chacun des cas de figure*)
       solutions (1., 1.,1.) (* pas de solutions réelle*);
       solutions (1., 0. ,-1.) (* 2 racines différentes *)
       solutions (1., 2. ,1.) (*  racine double*)
       (* test erroné *)
       solutions (0., 1.3 ,-1.) (*echec*)
*)

let solutions (a,b,c) =
     if a = 0.0 then failwith "solutions : premier degre"
     else  let delta = b*. b -. 4.0 *. a *. c in
             if delta < 0.0 then 0
             else if delta=0.0 then 1
                  else 2;;
```

Exercice 2 (**couples**)

Le rationnel $\frac{p}{q}$ sera représenté par le couple (p, q) de type `int*int`.

1. Ecrire la fonction `inverse_ratio` qui calcule l'inverse d'un nombre rationnel.

```
(* Interface inverse_ratio
type : int * int -> int*int
arguments (p,q)
pre : p<>0 et (p,q) rationnel valide
post inverse_rati0 (p,q)=(q,p)
raises : Failure "rationel nul" si p=0
tests : inverse_ratio (4,5) (*test nominal*)
        (*tests erronés : on viloe la precondition *)
        inverse_ratio (0,3)
        inverse_ration (3,0)
*)
let inverse_ratio (p,q) = if p=0 then failwith "rationel nul"
                          else (q,p);;
```

Remarquons le type inféré : il est plus général que type attendu. Tant mieux ! On supposera (pré condition) que q est non nul sinon, le rationnel argument n'est pas correct.

2. Ecrire la fonction qui réalise l'addition de 2 nombres rationnels. On ne cherchera pas à simplifier le rationnel résultat.

```
(* Interface inverse_ratio
type : int * int -> int*int
arguments (p,q)
pre : (p1,q1) et (p2, q2) rationnels valides
post retourne la somme des 2 rationnels
*)
let plus_ratio ((p1,q1),(p2,q2)) = (p1*q2 + p2*q1, q1*q2);;
```

Exercice 3 (**Fonctions récursives simples**)

1. Ecrire une fonction récursive `u` telle que `u n` calcule le nième terme de la suite $(u_n)_n$ définie par : $u_0 = 1$ et $u_n = sqrt(u_{n-1} + 2), n > 0$. La fonction *sqrt* de type `float -> float` existe et calcule la racine carrée.

```
let rec u n = if n=0 then 1. else sqrt (u (n-1) +. 2.);;
```

Ici nous avons supposé que la fonction n'était définie que sur les entiers positifs. Faire un test avec un nombre négatif (cas d'erreur)

2. Ecrire une fonction récursive `somme` qui calcule la somme des n premiers entiers naturels : $somme\ n = \Sigma_{i=1}^{i=n} i$

```
let rec somme n = if n = 0 then 0 else n + (somme (n-1));;
```

3. Ecrire une fonction récursive `carre` qui calcule la somme des n premiers carrés : $carre\ n = \Sigma_{i=1}^{i=n} i^2$

```
let rec carre n = if n = 0 then 0 else (n*n) + (carre (n-1));;
```

4. Ecrire la fonction `puissance` en utilisant une méthode dichotomique, c'est-à-dire en utilisant les résultats suivants :
   $a^{2k} = (a^2)^k$
   $a^{2k+1} = (a^2)^k * a$
   On supposera a et n entiers naturels.

```
let rec dicho (a,n) =
   if n=0 then 1
   else let y = dicho (a*a, (n / 2)) in
           if n mod 2 = 0 then  y
           else  y*a;;
```

   Utilité du let ... in local : on évite ainsi de recalculer 2 fois la même chose.

5. Reprendre la fonction `puissance` en utilisant les résultats suivants :
   $a^{2k} = (a^k)^2$
   $a^{2k+1} = (a^k)^2 * a$
   On supposera a et n entiers naturels.

```
let rec dicho (a,n) =
   if n=0 then 1
   else let y = dicho (a, (n / 2)) in
           if n mod 2 = 0 then  y*y
           else  y*y*a;;
```

6. Ecrire une fonction récursive `somme_puis` qui calcule $\Sigma_{i=0}^{i=n} x^i$, avec n un naturel quelconque et x un entier quelconque. Modifier l'écriture de cette fonction de manière à optimiser les calculs.

```
let rec somme_puis (a,n) =
   if n=0 then 1
   else somme_puis (a,n-1) + dicho(a,n)
```

   Pour optimiser les calculs, l'idée est de se souvenir de la dernière puissance calculée car à l'étape suivante il suffira de multiplier ce terme par a pour obtenir le nouveau terme. Cela revient à calculer simultanément 2 suites récurrentes définies par :

$$sp_0 = 1 \quad t_0 = 1$$

$$sp_n = sp_{n-1} + t_n \quad t_n = t_{n-1} * a$$

```
let rec sp_t (a,n) =
        if n=0 then (1,1)
        else let (sp', t') = sp_t (a,n-1) in
               let tn=t'*a in
               (sp'+tn, tn);;

let somme_puis (a,n) = let (r,_) = sp_t (a,n) in r;;
```

Exercice 4 (**Listes**)

1. Ecrire une fonction qui compte le nombre d'éléments d'une liste

```
let rec nb l = match l with
     [] -> 0
   | _::r -> 1 + (nb r);;
```

La fonction est polymorphe.

On peut aussi écrire une fonction qui utilise List.hd et List.tl. Mais le style est moins élégant.

2. Ecrire une fonction qui compte le nombre d'éléments pairs d'une liste d'entiers

```
let rec pairs l = match l with
     [] -> 0
   | x::r -> if x mod 2 = 0 then 1 + (pairs r)
             else pairs r;;
```

3. Ecrire la fonction `somme` qui fait la somme des nombres d'une liste de flottants.

```
let rec somme l = match l with
     [] -> 0.
   | x::r -> x +. (somme r);;
```

4. Ecrire la fonction `moyenne` qui fait la moyenne arithmétique des nombres d'une liste de flottants. Ecrire une version naive de la fonction et une version plus optimale (penser à calculer somme et nombre d'éléments en même temps). La fonction `float_of_int` convertit un entier en le flottant correspondant : ainsi l'expression `(float_of_int 2) +. 3.5` est une expression bien typée.

Solution naive et peu efficace car elle parcourt 2 fois la liste

```
let rec moyenne l = if l = [] then  failwith "liste vide"
                    else  (somme l) /. (float_of_int (nb l));;
```

Solution plus judicieuse qui utilise une fonction auxiliaire qui calcule simultanément la somme et le nombre.

```
let rec som_nb l = match l with
                      [] -> (0., 0)
                   |  x::r -> let (a,b) = som_nb r in
                              ((x+.a), (1 + b));;

let moyenne l = if l = [] then  failwith "liste vide"
                else  let (m,n) = som_nb l in m/.(float_of_int n);;
```

5. Ecrire la fonction `intervalle` : `intervalle (n,m)` retourne la liste des entiers compris entre `n` et `m` inclus. Si `n` est strictement supérieur à `m`, le résultat calculé est la liste vide.
Généraliser cette fonction en introduisant un pas (supposé positif non nul).

```
let rec intervalle (n,m) = if n > m then []
                           else n :: (intervalle (n+1,m));;
```

Cas de tests : `intervalle (3,4);; intervalle (5,5);; intervalle (5,4);;`. *Le 2ème cas ne découle pas directement de la spécification mais plutôt d'un cas aux limites.*

```
let rec intervalle (n,m,p) = if n > m then []
                             else n :: (intervalle (n+p,m,p));;

(*Tests dans la limite de la précondition*)
intervalle (5,4,2);;
intervalle (1,4,2);;
intervalle (1,4,3);;
intervalle (1,4,5);;
```

6. Ecrire les fonctions `appartient` et `place` : `appartient` teste l'appartenance d'un élément à une liste et `place` donne la position d'un élément dans une liste (0 si pas dans la liste, $n > 0$ si l'élément est le $n$ième)

```
#let rec appartient (e,l)  =
           match l with  [] -> false
                     | x::r ->  if x = e then true
                                else  appartient (e,r);;

#(* interface place
args e,l
pre aucune
post retourne 0 si e n'est pas dans l, retourne la place de la 1ere
occurrence de e dans l (le 1er a la place 1)
tests   place (3, [4;5;3]) place (4, [4;5;4;3]) place (2, [4;5;3])
*)
let rec place (e,l)  =
match l with [] -> 0
| x::r ->  if x = e then 1
           else  let p = place (e,r) in
                 if p = 0 then 0 else 1+p;;

# place (3, [4;5;3]);;
- : int = 3
# place (1,[4;5]);;
- : int = 0
```

Pour éviter le test `p=0`, on peut vérifier au préalable que l'élément recherché est dans la liste. Mais cela demande un parcours de la liste.

La fonction `place` sera plus élégamment écrite en utilisant le mécanisme des exceptions.

Si on modifie la spécification de la liste : échec si l'élément n'est pas dans la liste

```
(* interface place
args e,l
pre e est dans l
post retourne la place de la 1ere
occurrence de e dans l (le 1er a la place 1)
raises échec avec Failure "place : élément absent" si e n'est pas dans l
tests   place (3, [4;5;3]) place (4, [4;5;4;3]) place (2, [4;5;3])
*)
let rec place (e,l)  =
match l with [] -> failwith "place : élément absent"
```

```
| x::r ->  if x = e then 1
           else  1 + (place (e,r));;
```

7. Ecrire une fonction `min` qui retourne le plus petit entier d'une liste d'entiers. Peut on l'utiliser avec une liste de chînes de caractères ?

8. Ecrire une fonction qui élimine les doublons d'une liste (tout élément ne peut alors figurer qu'une seule fois dans la liste résultat)

```
let rec doublons l = match l with [] -> []
                        | x::r -> if appartient x r then doublons r
                                  else x::(doublons r);;
```

On peut utiliser directement `List.mem` *qui est la fonction prédéfinie OCaml pour appartenance d'un élément à une liste. Elle s'appelle mem et se trouve dans le module List, d'où l'écriture List.mem*

9. Faire l'intersection sans doublons de 2 listes sans doublons.

```
let rec inter (l1,l2) = match l1 with [] -> []
                        | x::r -> if appartient (x,l2) then x::(inter (r, l2))
                                  else inter (r,l2);;
```

Ce programme atteint bien son objectif uniquement si les 2 listes en argument sont des listes sans doublons. Si la précondition n'est pas vérifiée, on ne garantit rien quant au résultat.

10. Faire l'union sans doublons de 2 listes sans doublons.

```
let rec union (l1,l2) = match l1 with
          [] -> l2
      |   x::r -> if appartient(x,l2) then union(r,l2)
                  else x::(union (r,l2));;
```

Exercice 5 (**Listes triées**)

On désire maintenant manipuler des listes triées dans l'ordre croissant par exemple.

1. Ecrire une fonction qui insére un élément dans une liste triée. Le résultat obtenu doit être une liste triée dans l'ordre croissant. Vous essayerez de donner pour cette fonction une postcondition la plus formelle possible (proche d'une formule de la logique des prédicats).

```
(* interface inserer
type : 'a *'a list -> 'a list
arg e , l
pre : l est une liste triée dans l'ordre croissant
post : retourne la  liste triee dans l'ordre croissant composée des éléments de l et de e
tests : inserer (1, [2; 3; 4]) (* [1;2;3;4]*), inserer (3, [2;  4]) (* [2;3;4]*),
inserer (3, [2; 3; 4]) (* [2;3;3;4]*),
inserer (5, [2; 3; 4]) (* [2;3;4;5]*)
*)

let rec inserer (e ,l) = match l with
             [] -> [e]
          |  x::r -> if e<x then e::l
                     else x::(inserer (e, r));;
```

On peut définir la notion de liste triée de la facon suivante :
pour tout i, j, si i¿j alors l'element de la liste l à la position i est ¿= à l'element de l à la position j.

2. Faire la concaténation de deux listes triées de manière à obtenir une nouvelle liste triée (on parle alors de fusion).

```
(* interface inserer
type : 'a list*'a list -> 'a list
arg  l1, l2
pre : l1 et l2 sont deux listes  triées dans l'ordre croissant
post : retourne la  liste triee dans l'ordre croissant composée des éléments de l1 et de l2
tests : fusion ([1;3;4;4],[2;5]) (*[1;2;3;4;4;5]*)
*)
  let rec fusion (l1, l2) = match (l1, l2) with
| ([], []) -> []
| ([], _) -> l2
| (_, []) -> l1
| (x1::l'1, x2::l'2) ->
       if x1 < x2
          then x1::(fusion (l'1, (x2::l'2)))
          else x2::(fusion ((x1::l'1), l'2));;
```