NFP119 : corrigé feuille d'exercices 2

María-Virginia Aponte

10 octobre 2013

Exercice 1

1. Testez cette fonction en Ocaml pour les appels suivants : `sommeN(0)`, `sommeN(-1)`, `sommeN(1)`, `sommeN(3)`. L'appel `SommeN(n)` calcule la somme des n premiers nombres positis $1+2+3+\ldots+n$ si n est positif, et 0 n est négatif ou nul.

```
# sommeN (0);;
- : int = 0

# sommeN (-1);;
- : int = 0

# sommeN 1;;
- : int = 1

# sommeN 3;;
- : int = 6
```

2. Déroulez manuellement ces appels. *Rappel* : un déroulement récursif peut se faire manuellement en remplaçant chaque appel à la fonction par son corps où le paramètre formel est remplacé par le paramètre de l'appel, puis en réalisant les opérations correspondantes (en mimant l'exécution). Exemples pour la factorielle dans les transparents du cours.

```
sommeN (-1)=
   = if -1<=0 then 0 else ...
   = 0

sommeN(0) =
   = if 0<=0 then 0 else ...
   = 0

sommeN(3) =
   = if 3<=0 then 0 else 3 + sommeN(2)
   = 3 + sommeN (2)
   = 3 + (if 2<=0 then 0 else 2 + sommeN (2-1))
   = 3 + (2 + sommeN (1))
   = 3 + (2 + (if 1<=0 then 0 else 1 + sommeN (1-1))
   = 3 + 2 + (1 + sommeN (0))
   = 3 + 2 + 1 + sommeN(0)
   = 3 + 2 + 1 + (if 0<=0 then 0 else 0 + sommeN (-1))
   = 3 + 2 + 1 + 0
   = 6
```

3.

4. Cette fonction termine-t-elle dans tous les cas ? Pouvez vous dire pourquoi ?

   Oui.
   Si $n \leq 0$ elle termine car il n'y a plus d'appel récursif.
   Si $n > 0$ elle termine car chaque appel récursif se fait sur le paramètre $n - 1$ qui est plus petit que n.
   Le cas de base où $n = 0$ est atteint en un n ombre fini d'étapes.

Exercice 2

Terminaison
Considérez les définitions de fonctions suivantes. Ces fonctions terminent-elles toujours ? Donnez un déroulement récursif manuel pour les appels donnés plus bas, puis testez ces appels dans la boucle intéractive d'Ocaml.

1. ```
   # let rec fact n =
        if n=1 then 1 else n*fact(n-1);;
   ```
   Testez cette fonction avec les appels (fact 2), (fact 0) et fact (-1). Quel est le problème ? Donnez une correction simple.

   Solution : Cette fonction termine pour les arguments positifis, mais ne termine pas si n est négatif ou nul.

   ```
   fact(0)=
    =  if 0=1 then 1 else 0 * fact(0-1)
    =  0 * fact(-1)
    =  0 * (if -1=1 then 1 else (-1)* fact(-2))
    =  0 * (-1)* fact(-2)
    =  0 * (-1) * fact(-2)
    =  0 * (-1) * (-2) * fact(-3) * ...
   ```

   Une correction simple, consiste à changer la condition du cas de base de manière à capturer également ces deux cas-là. Cela donne :

   ```
   # let rec fact n =
        if n<=1 then 1 else n*fact(n-1);;
   ```

2. ```
   # let rec fact n =
        if n<=1 then 1 else n*fact(n+1);;
   ```
   Testez cette fonction avec les appels (fact 0) et (fact 1). Quel est le problème ? Donnez une correction simple.

   Solution : Cette fonction ne termine que si l'argument passé est plus petit ou égal à 1 ; S'il est plus grand, la fonction boucle dans les appels récursifs : on fait un appel récursif pour un argument à chaque fois plus grand. Comme le cas de base est 1 ou plus petit, on ne s'arrête jamais.

   ```
   fact 2 ⇒
   if 2<=1  then  1  else  2*fact (2+1) ⇒
   2 * (fact(3)) ⇒
   2 * (if 3<=1 then 1 else 3*factorielle (3+1)) ⇒
   2 * (3*factorielle (4)) ⇒
   3 * (2* (if 4<=1 then 1 else ...) ⇒
   ...
   ```

3. `# let rec f n = f(n-1);;`
   Testez cette fonction avec l'appel f 0. Quel est le problème ?

   Solution : Cette fonction ne termine jamais car elle n'a qu'un cas d'appel récursif et pas de cas de base.

4. ```
   # let rec g n =
       if n<=0 then g(n-1) else g(n+1);;
   ```
   Testez cette fonction avec les appels g 0 et g 1. Quel est le problème ?

   Solution : Cette fonction ne termine jamais car elle n'a que des cas d'appels récursifs et pas de cas de base.

```
g 0 =
 =  if 0<=0 then g(0-1) else g(0+1)
 =  g(-1)
 =  if -1<=0 then g(-1-1) else g(-1+1)
 =  g(-2)
 =  g(-3)
 = ...

g 1 =
 =  if 1<=0 then g(1-1) else g(1+1)
 =  g(2)
 =  g(3)
 = ...
```

Exercice 3

1. Donnez une définition récursive dans un style mathématique pour cette fonction.

$$x^b = \begin{cases} 1 & \text{si } b = 0 \\ x \times x^{(b-1)} & \text{si } b > 0 \end{cases}$$

2. Ecrivez cette fonction en Ocaml.

```
# let rec puissance x b =
      if b<=0 then 1 else x * puissance x (b-1);;
val puissance : int -> int -> int = <fun>
```

   Notez que l'on élargit le cas de base aux entiers plus petits ou égaux à 0. Pourquoi ?

3. Testez votre fonction. Termine-t-elle toujours ? Pourquoi ?

```
# puissance 3 0;;
- : int = 1

# puissance 3 (-1);;
- : int = 1

# puissance 3 2;;
- : int = 9
```

Oui, elle termine (par un arugment similaire à celui donné pour l'exercice 2).
```

Exercice 4

Question 1

Récursivité, programmation, fonctions d'ordre supérieur Considérez les fonction suivantes :

```
let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z');;

let rec motI (s,i) =
   if i<=0 then true
   else (lettre (s.[i])) && motI (s,(i-1));;
val motI : string * int -> bool = <fun>

# motI ("abcd456",3);;
- : bool = true

# motI ("abcd456",5);;
- : bool = false
```

1. Donnez un déroulement récursif de l'appel `motI ("abcd456",3)`. Testez en Ocaml les exemples proposés plus haut.

   ```
   motI ("abcd456",2) =
    = if 2<=0 then true else (lettre (s.[2])) && motI (s,(1))
    = lettre (s.[2])) && motI(s,(1))
    = lettre ('c') && motI(s,(1))
    = true && motI(s,(1))
    = motI(s,(1))
    = (if 1<=0 then true else (lettre (s.[1])) && motI (s,(0)))
    = lettre (s.[1])) && motI (s,(0))
    = lettre ('b')) && motI (s,(0))
    = true && motI (s,(0))
    = motI (s,(0))
    = (if 0<=0 then true else ...)
    = true

   motI ("abcd456",5) =
    = if 5<=0 then true else (lettre (s.[5])) && motI (s,(4))
    = lettre (s.[5])) && motI(s,(4))
    = lettre ('5') && motI(s,(4))
    = false && motI(s,(4))
    = false
   ```

2. Donnez une définition mathématique de cette fonction. Que fait la fonction `motI` ? Quel est le rôle du paramètre i ?

   $$motI(s,i) = \begin{cases} true & \text{si } i \leq 0 \\ lettre(s.[i]) \wedge motI(s, i-1) & \text{si } i > 0 \end{cases}$$

   L'appel $motI(s,i)$ teste si la sous-chaîne de s comprise entre les indices $i \ldots 0$ est composée uniquement de lettres. Le parcours débute au caractère d'indice i et termine au caractère d'indice 0. Le rôle du paramètre i est de donner l'indice de départ du parcours récursif.

3. On souhaite écrire une fonction qui teste si une chaîne est composée uniquement de lettres. Proposez une solution qui incorporent les fonctions `lettre` et `motI`. Cette dernière possède 2 arguments mais un seul change pendant les appels récursifs. Utilisez le schéma récursif 2 décrit dans les transparents du cours.

```
let  est_mot s =
 let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z') in
 let rec motI i =
   if i<=0 then true
   else (lettre (s.[i])) && motI (i-1)
   in motI (String.length s -1)
val est_mot : string -> bool = <fun>

# est_mot "aBc";;
- : bool = true

# est_mot "6B7s";;
- : bool = false
```

4. Que fait la fonction suivante ? Comparez la avec la fonction que vous avez écrite. Comment éffectue-t-elle le parcours récursif de la chaîne ? Donnez un déroulement récursif pour étayer votre réponse.

```
let  est_mot s =
 let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z') in
 let rec parcours i =
   if i>= String.length s then true
   else (lettre (s.[i])) && parcours (i+1)
 in parcours 0;;
val est_mot : string -> bool = <fun>

# est_mot "abc";;
- : bool = true

# est_mot "687678";;
- : bool = false
```

Solution : La fonction `est_mot` teste si une chaîne de caractères est uniquement composée de lettres. Elle utilise la fonction locale récursive `parcours`, qui parcourt la chaîne du premier au dernier indice en testant si le caractère visité est une lettre. Contrairement à la fonction $motI$, le parcours débute ici au premier caractère, et c'est pourquoi le cas d'arrêt teste si `i>=String.length s`, et le cas récursif fait un appel sur l'indice suivant $(i + 1)$.

5. La directive `trace` permet de tracer tous les appels déclenchés par une fonction récursive. Cela ne marche pas pour les fonctions récursives locales. Nous allons donc définir `parcours` globalement afin de tester plusieurs appels. Attention : il faut déclarer la fonction à tracer par `# trace nom-fonction`, où l'on écrit explicitement un caractère `#`

```
# let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z') ;;
val lettre : char -> bool = <fun>

#  let rec parcours i =
       if i>= String.length s then true
       else (lettre (s.[i])) && parcours (i+1);;
    val parcours : int -> bool = <fun>


# #trace parcours;;
parcours is now traced.

# parcours 0;;
parcours <-- 0
parcours <-- 1
```

```
parcours <-- 2
parcours <-- 3
parcours --> true
parcours --> true
parcours --> true
parcours --> true
- : bool = true
```

6. Inspirez vous de l'écriture compacte de la fonction `palindrome` dans les transparents du cours afin de donner une version plus courte de la fonction `parcours` qui utilise des opérateurs logiques plutôt qu'une conditionnelle.

```
# let est_mot s=
    let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z') in
    let rec parcours i =
       i>= (String.length s) || lettre (s.[i]) && parcours (i+1)
    in parcours 0;;
val est_mot : string -> bool = <fun>
```

Question 2

Inspirez-vous de la question précédente pour écrire la fonction qui transforme une chaîne de caractères composée uniquement de chiffres en un nombre entier. *Indication* : vous pourrez convertir chaque caractère vers son correspondant en chiffre entier, puis multiplier celui-ci par la puissance de 10 correspondant à son indice dans la chaîne. La somme de toutes ces opérations correspondra alos au résultat final. Exemple : le résultat à envoyer pour "283" est obetnue par $2 \times 10^2 + 8 \times 10^1 + 3 \times 10^0 = 200 + 80 + 3 = 283$.

```
let  chaine_vers_nombre s =
 let chiffre c = if ('0' <= c && c<= '9') then Char.code c - Char.code '0'
                 else failwith "chaine_vers_nombre" in
 let rec parcourt i pdix  =
   if i<0 then  0
   else (chiffre(s.[i]))* pdix + parcourt (i-1) (pdix*10)
 in parcourt (String.length s-1) 1;;
val chaine_vers_nombre : string -> int = <fun>

#  chaine_vers_nombre  "0";;
- : int = 0

# chaine_vers_nombre "187";;
- : int = 187
```

Question 3

1. Ecrivez une fonction `testeChaine` qui prend une chaîne s et une fonction de test p et teste si tous les caractères de s satisfont le test p. Quel est le type de votre fonction ?

```
# let testeChaine s p=
   let rec testeRec i =
       i>= (String.length s) || p (s.[i]) && testeRec (i+1)
   in testeRec 0;;
val testeChaine : string -> (char -> bool) -> bool = <fun>
```

2. Réecrivez la fonction `est_mot` par un simple appel à la fonction `testeChaine`.

```
# let lettre c = ('a' <= c && c<= 'z') or ('A'<=c && c<= 'Z');;
val lettre : char -> bool = <fun>

# testeChaine "bonjour" lettre;;
- : bool = true

# testeChaine "bon67" lettre;;
- : bool = false

# let est_mot s = testeChaine s lettre;;
val est_mot : string -> bool = <fun>

# est_mot "bon67";;
- : bool = false

# est_mot "bonjour";;
- : bool = true
```

3. Ecrivez une fonction `est_nombre` qui teste si une chaîne ne contient que des nombres par un appel à `testeChaine`.

```
# let nombre c = '0' <= c && c<= '9';;
val nombre : char -> bool = <fun>

# let est_nombre s = testeChaine s nombre;;
val est_nombre : string -> bool = <fun>

# est_nombre "8750";;
- : bool = true
```

Exercice 5

Polymorphisme, typage.

Pour les fonctions suivantes, tentez de découvrir leur type, puis comparez votre réponse avec celle donnée par Ocaml.

```
# let f(x,y) = (y,x);;
val f : 'a * 'b -> 'b * 'a = <fun>

# let f(x,y) = x^y;;
val f : string * string -> string = <fun>

#  let f(x,y) = (x+y, x-y, x*y);;
val f : int * int -> int * int * int = <fun>
```

Ici, x et y ne peuvent pas être à la fois de type int et string.

```
# let f(x,y) = (x+y,  x^y);;
This expression has type int but is here used with type string
```

Ici, appliquer f équivaut à appliquer fst. La fonction f devient une soret d'alias pour fst. Elles ont donc des types identiques.

```
# let f x = fst x;;
val f : ('a * 'b) -> 'a  = <fun>
```

Ici, x est de type quelconque. Par ailleurs, fst est appliquée sur y, et donc celui-ci doit nécesairement avoir le type d'une paire.

```
# let f(x,y) = (x, fst y);;
val f : 'a * ('b * 'c) -> 'a * 'b = <fun>
```

Même raisonnement qu'avant mais cette fois sur x et y.

```
# let f(x,y) = (fst x, fst y);;
val f : ('a * 'b) * ('c * 'd) -> 'a * 'c = <fun>
```

Même raisonnement sur x et y. De plus, les premières composantes de x et y sont nécessairement de type int, ainsi que le résultat de la fonction.

```
# let f(x,y) = fst x +  fst y;;
val f : (int * 'a) * (int * 'b) -> int = <fun>

# let g x y = (x,y);;
val g : 'a -> 'b -> 'a * 'b = <fun>

# let g x y = (x,y,x&&y);;
val g : bool -> bool -> bool * bool * bool = <fun>
```

Exercice 6

Filtrage.

Lorsqu'il y a erreur, expliquez les messages affcihés par Ocaml, et proposez une solution. Pour les fonctions sans erreur, testez-les sur plusieurs exemplez et expliquez ce qu'elles font.

Solution : La fonction `voyelle` teste si un caractère est une voyelle.

```
# voyelle 'b';;
- : bool = false
# voyelle 'i';;
- : bool = true
```

La fonction `opArith` est définie de sorte que le premier cas filtre tous les argument possibles de la fonction. Du coup, elle est polymorphe et accepte n'importe quel type d'argument. De plus, elle répond `false` toujours. La solution est d'inverser les deux filtres :

```
#let  opArith c =
 match c
 with    _ -> false;;
  | '+'| '-' | '*' | '/'  -> true
val opArith : 'a -> bool = <fun>

# opArith 1;;
- : bool = false
# opArith '+';;
- : bool = false

(* Correction *)

#let  opArith c =
 match c
 with  '+'| '-' | '*' | '/'  -> true
```

```
  |  _ -> false;;
val opArith : 'a -> bool = <fun>

# opArith 1;;
This expression has type int but is here used with type char
#  opArith '+';;
- : bool = true

# opArith 1;;
- : bool = false
# opArith '+';;
- : bool = false
```

La fonction `sommeTriple` compare son entrée avec deux cas de filtres de structure différente, et donc, dont les types sont différents. L'erreur signalé est une erreur de typage. La solution est de changer le deuxième filtre pour que ce soit un triplet.

```
# let sommeTriple a =
   match a
   with (x,y,z) -> x+y+z
    |   (0,0,0)   -> -1;;
Warning: this match case is unused.
val sommeTriple : int * int * int -> int = <fun>

# sommeTriple (0,0,0);;
- : int = 0
```

La fonction `sommeTriple` est maintenant bien typée, mais elle ne fait pas ce que l'on veut : le dernier cas n'est jamais utilisé, car le premier filtre capture toutes les entrées. On inverse leur ordre :

```
#let sommeTriple a =
 match a
 with  (0 ,0,0)   -> -1
  |   (x,y,z) -> x+y+z;;
val sommeTripleBis : int * int * int -> int = <fun>

#sommeTriple (0,0,0);;
- : int = -1
```

La fonction `sommeTripleBis` est mal typée car elle renvoie des valeurs de types différents dans ces deux cas. La fonction `reponse` ne repertorie pas tous les cas possibles de son argument. L'erreur signalé par Ocaml nous dit que le filtrage est non exhaustif. Du coup, la fonction n'est pas définie pour tous les cas : elle échoue pour les cas non capturés par les filtres donnés.

```
# let reponse r =
  match r
  with"oui" -> trueue
  |     "Oui" -> true
  |     "OUI" -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
""
val reponse : string -> bool = <fun>

val reponse : string -> bool = <fun>
# reponse "oui";;
- : bool = true
```

```
# reponse "non";;
Exception: Match_failure ("", 2, 0).
```

La solution est de compléter le filtrage :

```
# let reponse r =
  match r
  with"oui" -> true
  |     "Oui" -> true
  |     "OUI" -> true
  |     _ -> false;;
val reponse : string -> bool = <fun>

# reponse "oui";;
- : bool = true
#  reponse "non";;
- : bool = false
```

Exercice 7

Définir un type enregistrement `date` pour modéliser une date composée d'un numéro de jour, d'un numéro de mois et d'une année.

```
type date = {jour:int; mois:int; annee: int};;
type date = { jour : int; mois : int; annee : int; }
```

Ecrivez ensuite :

1. Un exemple de date dans la variable `aujourd'hui`.
2. Une fonction `bissextile` qui teste si une année est bissextile.

   ```
   # let bissextile a =
     (a mod 4 =0) && (not (a mod 100 = 0) or (a mod 400 = 0));;
   val bissextile : int -> bool = <fun>

   # bissextile 2000;;
   - : bool = true

   # bissextile 1900;;
   - : bool = false

   # bissextile 2004;;
   - : bool = true
   ```

3. Une fonction `joursMois` qui calcule le nombre de jours dans un mois pour une année donnée.

   ```
   # let joursMois m a =
   match m
   with 4 | 6 | 9 | 11 -> 30
    |   2 -> if bissextile a then 29 else 28
    |   _ -> 31

   val joursMois : int -> int -> int = <fun>

   # joursMois 2 2004;;
   - : int = 29
   ```

```
# joursMois 2 2005;;
- : int = 28

# joursMois 3 2006;;
- : int = 31
```

4. En utilisant les deux fonctions précédentes, écrire une fonction `lendemain` qui étant donné une date supposée correcte calcule la date du lendemain.

```
# let lendemain d =
    if d.jour < joursMois d.mois d.annee
            then {jour= d.jour+1; mois=d.mois; annee = d.annee}
    else if d.mois < 12
            then {jour= 1; mois=d.mois+1; annee = d.annee}
    else {jour= 1; mois= 1; annee = d.annee+1}
val lendemain : date -> date = <fun>

#let aujourd'hui = {jour=27; mois=2; annee = 2006};;
val aujourd'hui : date = {jour = 27; mois = 2; annee = 2006}

# lendemain aujourd'hui;;
- : date = {jour = 28; mois = 2; annee = 2006}
```

Exercice 8

Définissez un type `employe` caractérisé par un nom, un salaire, et une date d'entrée dans l'entreprise.

1. Ecrivez une fonction qui en prenant deux employés, renvoyer celui dont le salaire est le plus élévé.
2. Ecrivez une fonction pour calculer l'ancienneté d'un employé en mois et années (par exemple 2 mois, ou 1 an et 2 mois).

```
#type employe = {nom: string; salaire: float; dateArrivee: date};;

#type duree = {n_mois: int; n_annees: int};;

#let anciennete aujourd'hui {dateArrivee = e} =
 let soustrait_dates d1 d2 =
   if d1.annee = d2.annee then {n_mois = d1.mois - d2.mois; n_annees = 0}
   else if d1.mois = d2.mois
               then {n_mois = 0; n_annees = d1.annee-d2.annee}
   else if d1.mois > d2.mois
               then {n_mois = d1.mois - d2.mois; n_annees = d1.annee-d2.annee}
   else {n_mois = (12-d2.mois)+ d1.mois; n_annees = d1.annee-d2.annee-1}
 in soustrait_dates aujourd'hui e;;

val anciennete : date -> employe -> duree = <fun>

# let sep2000 = {jour =5; mois=9; annee = 2000};;
val sep2000 : date = {jour = 5; mois = 9; annee = 2000}

# let martin = {nom = "Martin"; salaire = 2200.50; dateArrivee=sep2000};;
val martin : employe =
  {nom = "Martin"; salaire = 2200.5;
   dateArrivee = {jour = 5; mois = 9; annee = 2000}}

# anciennete aujourd'hui martin;;
```

```
- : duree = {n_mois = 5; n_annees = 5}


# let plus_paye e1 e2 =
 if e1.salaire > e2.salaire then e1 else e2;;
```