# Corrigé du partiel de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

1er avril 2003

1. *Écrire une fonction qui, étant donné $n \geq 1$, calcule $u_n$, où la suite $(u_n)_{n\geq 1}$ est définie par*

$$u_1 = 1$$
$$u_{n+1} = \begin{cases} u_n/2 + 1 & \text{si } u_n \text{ est pair} \\ u_n + 17 & \text{sinon} \end{cases}$$

*On prendra garde à ne pas faire un même calcul plusieurs fois.*
Réponse :

```
let rec u = function
  1 -> 1
| n -> let x = u (n-1)
       in if x mod 2 = 0 then x/2 + 1 else x + 17
```

2. *Donner le type des expressions suivantes :*
   - `fun x -> x +. 3.0`
   - `fun (x,y) -> x or y`
   - `fun x -> let f y = y + x in f x`
   - `List.map (fun x -> not x)`
   - `fun f g -> (fun x -> f (g x))`

   Réponse :

```
# fun x -> x +. 3.0;;
- : float -> float = <fun>
# fun (x,y) -> x || y;;
- : bool * bool -> bool = <fun>
# fun x -> let f y = y + x in f x;;
- : int -> int = <fun>
# List.map (fun x -> not x);;
- : bool list -> bool list = <fun>
# fun f g -> (fun x -> f(g(x)));;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

   La troisième valeur est une fonction qui prend un `x` et le passe à la fonction `f`. Le type de `x` est donc le domaine de `f`. Soit $\alpha$ ce type. La fonction `f` prend un `y` (de type $\alpha$, donc) et l'ajoute à `x`. Or l'opérateur `(`

`+ )` a pour type `int` → `int` → `int`. Par conséquent `x` et `y` sont de type `int`, en particulier $\alpha =$ `int`, et le type de `f` est `int` → `int`. Donc `f(x)` a pour type `int`. Finalement, la fonction proposée a pour type `int` → `int`.
La quatrième expression repose sur `List.map` qui a pour type $(\alpha \to \beta) \to \alpha$ `list` → $\beta$ `list`. Or on l'évalue partiellement en son premier argument avec une fonction dont le type est `bool` → `bool`. Donc $\alpha = \beta =$ `bool`. Il reste donc finalement le type `bool list` → `bool list`.

3. *On désire manipuler symboliquement des expressions formées à partir des éléments suivants :*
   — *la constante $\pi$ ;*
   — *des variables $x$, $y$, etc. ;*
   — *un opérateur binaire $+$ ;*
   — *un ensemble fixe de fonctions : sin, cos, et tan.*

   *Questions :*
   — *Définir un type* `expression` *pour représenter ces expressions (les noms de variable peuvent être représentés par des chaînes de caractères) ;*

   Une expression est soit la constante $\pi$, soit une variable, soit $+$, soit une fonction trigonométrique. Il faut donc définir un type somme (suite de « ou »). On définit donc un constructeur pour chaque type d'expression, par exemple : `Pi`, `Var`, `Plus`, `Sin`, `Cos` et `Tan`. Ensuite il faut déterminer quelle information supplémentaire chaque constructeur doit porter pour modéliser complètement le concept correspondant. En ce qui concerne `Pi`, tout est dit : un constructeur constant (c-à-d. sans arguments) modélise bien une constante mathématique. Une variable cependant a un nom, et l'énoncé du sujet proposait le type `string` pour capturer cela, donc il faudra définir `Var of string`. Le constructeur `Plus` modélise une fonction mathématique qui opère sur deux arguments. Puisqu'on souhaite être le plus général possible et ne pas créer de types supplémentaires, le type qui s'impose pour les arguments est bien entendu `expression` lui-même (ce type est donc récursif) : `Plus of expression * expression`. Les fonction trigonométriques n'opèrent que sur un seul argument, donc on définira au final :

```
type expression =
  Pi
| Var of string
| Plus of expression * expression
| Sin of expression
| Cos of expression
| Tan of expression
```

   — *Donner la valeur Caml correspondant à l'expression* $\cos x + \tan y$.

   Réponse :
   `Plus ((Cos (Var "x")), (Tan (Var "y")))`

   — *Écrire une fonction* `eval` *qui calcule la valeur flottante d'une expression dans un environnement donné. On rappelle qu'un environnement*

*est une liste de paires (nom de variable, valeur) qui à chaque variable associe une valeur. La fonction prédéfinie* `List.assoc` *sera utile pour accéder aisément aux contenu de l'environnement. (On précise que les fonctions* `sin`*,* `cos` *et* `tan` *sont prédéfinies en Caml mais la constante* $\pi$ *ne l'est pas : on utilisera la fonction arc-cosinus* `acos`*.)*

Réponse :

```
# let rec eval env = function
    Pi            -> acos (-1.0)
  | Var (x)       -> List.assoc x env
  | Plus (e1, e2) -> eval env e1 +. eval env e2
  | Sin (e)       -> sin (eval env e)
  | Cos (e)       -> cos (eval env e)
  | Tan (e)       -> tan (eval env e)
  ;;
val eval : (string * float) list -> expression -> float = <fun>
```

4. *On désire manipuler des polynômes à une variable et à coefficients entiers. On représentera un polynôme par une liste de couples (coefficient, degré), triée par ordre décroissant de degré ; chaque couple représente un monôme. Par exemple,* $X^3 - X + 1$ *sera représenté par la liste* `[(1,3); (-1,1); (1,0)]`*. Écrire une fonction qui effectue la somme de deux polynômes. On prendra garde à respecter l'ordre des monômes et à ne pas créer plusieurs monômes de même degré dans le résultat.*

   Réponse :

```
let rec ajout p1 p2 =
  match (p1, p2) with
    ([], _) -> p2
  | (_, []) -> p1
  | ((c1,d1)::r1, (c2,d2)::r2) ->
      if d1 = d2
      then (c1+c2,d1)::(ajout r1 r2)
      else if d1 > d2
           then (c1,d1)::(ajout r1 p2)
           else (c2,d2)::(ajout p1 r2)
```

   On remarque qu'on destructure les monômes (sous forme de paires) dans le motif pour ensuite parfois reconstruire les mêmes monômes (à droite des flèches) si `d1 > d2` ou `d1 < d2`. On peut alors légèrement améliorer les performances en liant *dans le filtre* la paire à un nom, et réutiliser ce nom à droite des flèches (c-à-d. dans le calcul) :

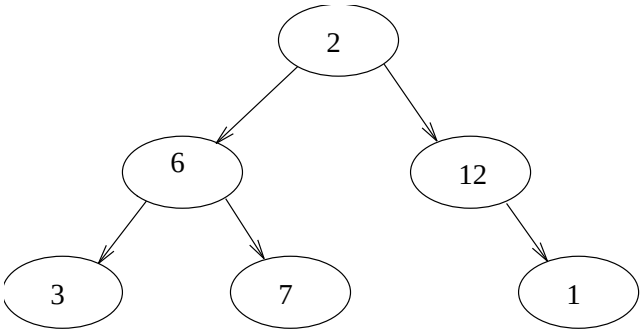
```
let rec ajout p1 p2 =
  match (p1, p2) with
    ([], _) -> p2
  | (_, []) -> p1
  | (((c1,d1) as m1)::r1, ((c2,d2) as m2)::r2) ->
      if d1 = d2
      then (c1+c2,d1)::(ajout r1 r2)
      else if d1 > d2
```

```
          then m1::(ajout r1 p2)
          else m2::(ajout p1 r2)
```

Ainsi les monômes `m1` et `m2` ne sont jamais reconstruits.
Pour éviter de construire des monômes dont le coefficient est nul (par `c1+c2`) on ajoute une conditionnelle :

```
let rec ajout p1 p2 =
  match (p1, p2) with
    ([], _) -> p2
  | (_, []) -> p1
  | (((c1,d1) as m1)::r1, ((c2,d2) as m2)::r2) ->
      if d1 = d2
      then if c1 + c2 = 0
           then ajout r1 r2
           else (c1+c2,d1)::(ajout r1 r2)
      else if d1 > d2
           then m1::(ajout r1 p2)
           else m2::(ajout p1 r2)
```

5. *Un arbre binaire est un arbre où chaque nœud a au plus deux fils, un fils gauche et un fils droit. De plus, chaque nœud est annoté par une étiquette, qui est une valeur* a priori *quelconque. Voici un exemple d'arbre annoté par des étiquettes de type entier :*




*Le type des arbres binaires est défini par :*

```
type 'a arbre =
  Rien
| Noeud of 'a * ('a arbre) * ('a arbre)
```

*Ce type est* paramétré *par la variable de type* `'a`*, ce qui permet de définir le type des arbres indépendamment du type des étiquettes. On pourra ensuite utiliser le type* `int arbre` *lorsque l'on veut annoter chaque nœud par un entier, etc.*

— *Écrire une fonction qui calcule la hauteur d'un arbre.*

```
let rec hauteur = function
  Rien -> 0
| Noeud (_, gauche, droit) ->
```

```
    1 + max (hauteur gauche) (hauteur droit)
```

La fonction `max` est polymorphe et prédéfinie, mais dans le cas précis (elle s'applique à des entiers naturels) elle pourrait être simplement définie par

```
let max a b = if a < b then b else a
```

— *Écrire une fonction qui calcule la taille d'un arbre (c'est-à-dire le nombre de nœuds).*

```
let rec taille = function
  Rien -> 0
| Noeud (_, gauche, droit) ->
    1 + taille (gauche) + taille (droit)
```

— *Écrire une fonction* `recherche`*, qui prend en argument une étiquette* `x` *et un arbre* `a`*, et indique si cette étiquette apparaît sur l'un des nœuds de* `a`*.*

```
let rec recherche x = function
  Rien -> false
| Noeud (elm, gauche, droit) ->
    elm = x || recherche x gauche || recherche x droit
```