TP1 – Les langages de programmation

À la découverte de Objective CAML

UV IDL

Correction

1 Une introduction

Exercice 1

La fonction `map` prend en argument une fonction `f` et une liste `l`. Elle doit retourner la liste du résultat des applications de `f` aux éléments de `l`.

▷ **Question 1.1 :**
Sans utiliser l'interpréteur, donner le type de la fonction `map`.

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

▷ **Question 1.2 :**
Construire la fonction `map` et vérifier son type.

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f t)
;;
```

Exercice 2

Même exercice que précédemment mais avec la fonction `iterate` qui prend en paramètre un entier n et une fonction f et calcule f^n $(f(f(\ldots f())))$.

```
int -> ('a -> 'a) -> 'a -> 'a
```

Il y deux méthodes pour réaliser la fonction `iterate`, une simple et une plus complexe.
Pour la simple, on exhibe le x en lequel sera calculé la fonction.

```
let rec iterate n f x =
  match n with
  | 0 -> x
  | _ -> f ((iterate (n-1) f) x)
;;
```

La version naïve suivante :

```
let rec iterate1 n f =
  match n with
  | 0 -> (function x -> x)
  | _ -> f (iterate1 (n-1) f)
;;
```

est du type `int -> (('a -> 'a) -> 'a -> 'a) -> 'a -> 'a` qui n'est alors pas utilisable avec une fonction f qui s'appliquerait à autre chose qu'une fonction (cos par exemple).
Le problème provient du fait qu'il va essayer d'appliquer f à l'identité (qui est une fonction) et donc imposer à la fonction f de prendre en paramètre une fonction.
Pour contourner cette contrainte, il faut définir une fonction permettant de composer deux fonctions.

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

let rec iterate2 n f =
  match n with
  | 0 -> (function x -> x)
  | _ -> compose f (iterate2 (n-1) f)
;;
```

2 Un peu plus de ocaml

Exercice 3 (*Des arbres*)

▷ **Question 3.1 :**
Construire un type récursif arbre binaire.

```
type 'a arbre_bin =
  | Empty
  | Node of 'a arbre_bin * 'a * 'a arbre_bin
```

▷ **Question 3.2 :**
Utiliser le type ainsi construit pour réaliser un arbre binaire de recherche entier. C'est-à-dire un arbre binaire ayant la propriété suivante : pour tous les nœuds, toutes

les valeurs du sous-arbre gauche sont inférieures à celle du nœud, et toutes celles du droit lui sont supérieures.

Pour cela construisez une fonction `ajout` qui ajoute un entier dans un arbre de recherche entier.

```
let rec ajout x = function
  | Empty  ->  Node(Empty, x, Empty)
  | Node(fg, r, fd)  ->
    if x < r then
      Node(ajout x fg, r, fd)
    else
      Node(fg, r, ajout x fd)
```

▷ **Question 3.3 :**
Utiliser la fonction définie à la fonction précédente pour réaliser une fonction de tri des listes d'entiers.

```
let rec list_of_arbre = function
  | Empty -> []
  | Node(fg, r, fd) -> (list_of_arbre fg) @ (r :: (list_of_arbre fd))

let rec arbre_of_list = function
   | []   ->  Empty
   | t::q  ->  ajout t (arbre_of_list q)

let tri x = list_of_arbre (arbre_of_list x)
```

3 Les expressions

Cette section contient uniquement un problème. Il s'agit de construire un évaluateur d'expression arithmétiques.

Exercice 4 (*Le calcul formel*)

▷ **Question 4.1 :**
Construire un type permettant de décrire des expressions arithmétiques basiques (des réels et les quatre opérateurs de base).

```
type expression =
  | Const of float
  | Sum of expression * expression (* e1 + e2 *)
  | Diff of expression * expression (* e1 - e2 *)
  | Prod of expression * expression (* e1 * e2 *)
```

```
  | Quot of expression * expression (* e1 / e2 *)
```

▷ **Question 4.2 :**
Construire une fonction `eval` qui permet d'évaluer une expression.

```
let rec eval exp =
  match exp with
  | Const c -> c
  | Sum(f, g) -> eval f +. eval g
  | Diff(f, g) -> eval f -. eval g
  | Prod(f, g) -> eval f *. eval g
  | Quot(f, g) -> eval f /. eval g
```

▷ **Question 4.3 :**
Étendre les expressions pour permettre l'utilisation de variables. Une variable sera une chaîne de caractères qui peut apparaître dans une expression. À l'évaluation, les variables auront une valeur.

```
type expression =
  | Const of float
  | Var of string
  | Sum of expression * expression (* e1 + e2 *)
  | Diff of expression * expression (* e1 - e2 *)
  | Prod of expression * expression (* e1 * e2 *)
  | Quot of expression * expression (* e1 / e2 *)
```

▷ **Question 4.4 :**
Modifier la fonction `eval` qui permet d'évaluer une expression dans un environnement. Un environnement est une liste d'association qui associe des valeurs à des variables. La rencontre d'une variable non définie doit conduire à un cas d'erreur.

```
exception Unbound_variable of string

let rec eval env exp =
  match exp with
  | Const c -> c
  | Var v -> (try List.assoc v env with Not_found -> raise(Unbound_variable v))
  | Sum(f, g) -> eval env f +. eval env g
  | Diff(f, g) -> eval env f -. eval env g
  | Prod(f, g) -> eval env f *. eval env g
  | Quot(f, g) -> eval env f /. eval env g
```

▷ **Question 4.5 :**
Écrire une fonction `string_of_expr`.

```
let rec string_of_expr exp =
  match exp with
  | Const c -> string_of_float c
  | Var v -> v
  | Sum(e1, e2) -> "(" ^(string_of_expr e1)^ " + " ^(string_of_expr e2)^ ")"
  | Diff(e1, e2) -> "(" ^(string_of_expr e1)^ " - " ^(string_of_expr e2)^ ")"
  | Prod(e1, e2) -> "(" ^(string_of_expr e1)^ " * " ^(string_of_expr e2)^ ")"
  | Quot(e1, e2) -> "(" ^(string_of_expr e1)^ " / " ^(string_of_expr e2)^ ")"
```

▷ **Question 4.6 :**
Construire une fonction `derive` qui dérive une expression par rapport à une variable passée en argument.

```
let rec deriv exp dv =
  match exp with
  | Const c -> Const 0.0
  | Var v -> if v = dv then Const 1.0 else Const 0.0
  | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
  | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
  | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
  | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)), Prod(g, g))
```

▷ **Question 4.7 :**
Construire une fonction `simplifie` qui permet de simplifier une expression en utilisant les règles suivantes :

$$\begin{cases} -0 = 0 \\ \forall e \quad e + 0 = 0 + e = e \\ \forall e \quad e \times 0 = 0 \times e = 0 \\ \forall e \quad e \times 1 = 1 \times e = e \end{cases}$$

```
let rec simplifie exp =
  match exp with
  | Const -0.0 -> Const 0.0
  | Sum(f, g) ->
      begin
        match (simplifie f,simplifie g) with
          | (Const 0.0, ng) -> ng
          | (nf, Const 0.0) -> nf
          | (nf,ng) -> Sum(nf,ng)
      end
  | Prod(f, g) ->
      begin
        match (simplifie f,simplifie g) with
          | (Const 0.0, _) -> Const 0.0
          | (_, Const 0.0) -> Const 0.0
          | (Const 1.0, ng) -> ng
```

```
        | (nf, Const 1.0) -> nf
        | (nf,ng) -> Prod(nf,ng)
    end
  | Diff(f, g) -> Diff(simplifie f, simplifie g)
  | Quot(f, g) -> Quot(simplifie f, simplifie g)
  | _ -> exp
```

4 S'il vous reste du temps ou pour travailler en plus

Exercice 5 (*Tarot*)

▷ **Question 5.1 :**
Définir un type permettant de décrire les cartes d'un jeu de tarot.

```
type couleur =
  | Pique
  | Coeur
  | Carreau
  | Trefle
type carte =
  | Roi of couleur
  | Dame of couleur
  | Cavalier of couleur
  | Valet of couleur
  | Petite_carte of couleur * int
  | Atout of int
  | Excuse
```

▷ **Question 5.2 :**
Définir une fonction `toutes_les_cartes` qui construit l'ensemble des cartes d'une couleur reçue en paramètre.

```
let toutes_les_cartes s =
  let rec les_autres  n =
    match n with
    | 0 -> []
    | _ -> (les_autres  (n-1)) @ [Petite_carte(s,n)]
  in [ Valet s; Cavalier s; Dame s; Roi s ] @ (les_autres 10)
```

▷ **Question 5.3 :**
Définir une fonction `string_of_cartes` qui construit une chaîne de caractères décrivant la carte reçue en paramètre.

```
let string_of_couleur = function
  | Pique   -> "pique"
  |  Carreau -> "carreau"
  |  Coeur   -> "coeur"
  |  Trefle  -> "trè fle "
let string_of_carte = function
  | Roi c         -> "roi  de " ^ (string_of_couleur c)
  |  Dame c        -> "dame de " ^ (string_of_couleur c)
  |  Valet c       -> "valet  de " ^ (string_of_couleur c)
  |  Cavalier c    -> " cavalier  de " ^ (string_of_couleur c)
  |  Petite_carte (c, n) -> (string_of_int n) ^ " de "^(string_of_couleur c)
  |  Atout n       -> (string_of_int n) ^ " d'atout"
  |  Excuse        -> "excuse"
```