Récursion, évaluation, filtrage, listes

Récursion

Exercice 1. Écrire une fonction OCaml « équivalente » à la fonction C

```
int sommeborne(int n, int (* f)(int))
{
  int x = 0, i=0;

  while ( i <= n)
    x += f(i++);

  return x;
}
```

Exercice 2. Définir une fonction OCaml `iter` dont le type est

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

telle que $iter\ n\ f\ x = f^n(x)$. Généraliser cette fonction à une fonction `fold` dont le type est

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

telle que $fold\ n\ f\ x = f(n, \ldots f(1, f(0, x))$.
Utiliser la fonction `fold` pour résoudre l'exercice précédent.

Exercice 3. La suite de Fibonacci peut être définie par une fonction OCaml comme il suit :

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

- Estimer la complexité de cette fonction.
- Proposer une définition de la même fonction plus efficace.
  (Suggestion : utiliser les produits Cartésiens.)

Stratégies d'évaluation

Exercice 4. Considérer les expressions et définitions suivantes :

```
(fun x -> fun y -> (x + y))(1 + 2)(fibo 0);;
(fun x -> x + x) 1 + 2;;
let f1 = fun f2 -> (fun x -> f2 x);;
let g = fun x -> x + 1;;
f1 g 2;;
```

Donner les étapes de l'évaluation par valeur des trois expressions précédentes. Évaluer ces mêmes expressions en utilisant la stratégie par nom.

Exercice 5. Considérer la fonction `sialorssinon`, définie comme il suit :

```
let sialorssinon x y z = if x then y else z ;;
```

- Comparer les évaluations par valeur des expressions
  - `sialorssinon true (3 + 4) (5 + 6)`,
  - `if true then (3 + 4) else (5 + 6)`,
- Proposer trois expressions `x`, `y`, et `z` telles que les évaluations par valeur de `sialorssinon x y z` et , `if x then y else z` ne donnent pas le même résultat.
- Que se passe-t'il si dans un langage fonctionnel dont la stratégie d'évaluation est par valeur, on ne possède ni des constructeurs comme `if ... then ... else ...` ?

Filtrage

Exercice 6. Modifier les deux définitions de la suite de Fibonacci en utilisant le filtrage par `function` et `match ... with`.

Listes

Exercice 7. Considérer la fonction

```
let rec longeur = function
     [] -> 0
   | t::q -> 1 + longeur q ;;
```

- Donner les étapes de l'évaluation par valeur de l'expression `longeur [1 ;2 ;3]`.
- Quel rapproche peut-on faire à la (définition de la) fonction `longeur` ? (Suggestion : examiner la complexité en temps et espace de cette fonction.)
- Proposer une amélioration de cette fonction.

Exercice 8.

algox.ml

```
 1 : let rec length = function
 2 :     [] -> 0
 3 :   | x::xs -> 1 + length(xs);;
 4 :
 5 : let rec split liste n =
 6 :   if n <= 0 then ([],liste )
 7 :   else
 8 :     match liste with
 9 :         [] -> ([],[])
10 :       | x::xs ->
11 :           let (l1,l2) = split xs (n-1) in
12 :             (x::l1,l2) ;;
13 :
14 : let rec merge liste1 liste2 lesseq =
15 :   match  (liste1,liste2) with
16 :       ([],_) -> liste2
17 :     | (_,[]) -> liste1
18 :     | (x::xs,y::ys) ->
19 :         if lesseq x y then
20 :           x::(merge xs liste2 lesseq)
21 :         else
22 :           y::(merge liste1 ys lesseq) ;;
23 :
24 : let rec algox lesseq = function
25 :     [] -> []
26 :   | [x] -> [x]
27 :   | liste  ->
28 :       let
29 :         moitie = length liste / 2
30 :       in
31 :       let
32 :         (l1,l2) = split liste moitie
33 :       in
34 :         merge ( algox lesseq l1) ( algox lesseq l2) lesseq ;;
```

Expliquer ce que fait le programme `algox.ml`

Exercice 9. Écrire une fonction OCaml, de type `int -> int`, qui calcule le n-ième nombre premier etant donné un nombre entier positif n. (Suggestion : on pourra construire d'abord une fonction `int -> int list` qui construit la liste des premiers n-nombres premiers.)