TDs de Logique. Feuille 4. Corrigé. Caml

Exercice 1.—

```
#type entier = Zero | Succ of entier;;
Type entier defined.
#let succ x = Succ x;;
succ : entier -> entier = <fun>
#let pred = function
           Zero -> Zero
         | Succ x -> x;;
pred : entier -> entier = <fun>
#let rec plus a y = match a with
     Zero -> y
   | Succ x -> Succ (plus x y);;
plus : entier -> entier -> entier = <fun>
#let rec mult a y  = match a with
     Zero  -> Zero
   | Succ x -> plus (mult x y) y;;
mult : entier -> entier -> entier = <fun>
#let rec fact = function
     Zero   ->  Succ Zero
   | Succ x ->  mult (Succ x) (fact x);;
fact : entier -> entier = <fun>
#let rec peano_to_int = function
     Zero  ->  0
  |  Succ x -> (peano_to_int x) +1;;
peano_to_int : entier -> int = <fun>
#let rec int_to_peano n =
      if n=0 then Zero
      else Succ (int_to_peano (n-1));;
int_to_peano : int -> entier = <fun>
```

Exercice 2.—

```
#let rec somme = function
       [] -> 0
   | x::l -> x + somme l;;
```

somme : int list -¿int = ¡fun¿

Exercice 3.—

```
#let rec list_it f l b =
   match l with
       [] -> b
   | x::reste -> f x (list_it f reste b);;
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

let somme l = list_it (prefix +) l 0;;
somme : int list -> int = <fun>
```

Exercice 4.—

```
#let rec mult x l =
    match l with
          [] -> []
  | y::reste -> (x*y)::(mult x reste);;
mult : int -> int list -> int list = <fun>
```

Exercice 5.—

```
#let rec map f = function
```

```
         [] -> []
       | x::l -> (f x)::map f l;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#let mult x l = map (function y->x*y) l;;
mult : int -> int list -> int list = <fun>
```

Exercice 6.—

```
#let compose g f = function x -> g (f x);;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Exercice 7.—

```
#let concat l m = l@m;;
- : int list = [1; 2; 3; 3; 4]
```

ou

```
#let rec concat l m =
    match l with
          [] -> m
  | x::reste ->  x::(concat reste m);;
concat : 'a list -> 'a list -> 'a list = <fun>
```

Exercice 8.—

```
#let rec rev  = function
          [] -> []
      | x::l ->  (rev l) @ [x];;
rev : 'a list -> 'a list = <fun>

#let rec renverse_auxi l accu =
    match l with
        [] -> accu
    | x::reste -> renverse_auxi reste (x::accu);;
renverse_auxi : 'a list -> 'a list -> 'a list = <fun>
#let renverse l = renverse_auxi l [];;
renverse : 'a list -> 'a list = <fun>
```

Exercice 9.—

```
#let rec fusion l1 l2 =
    match (l1,l2) with
       ([],[]) -> []
   |   (_, []) -> l1
   |   ([],_)  -> l2
   |(x::reste1,y::reste2) ->
    if x <=  y then x::(fusion reste1 l2)
               else y::(fusion l1 reste2);;
fusion : 'a list -> 'a list -> 'a list = <fun>
```

Exercice 10.—

```
#let rec inserer_tete a = function
            []->[]
        |l::l1 ->(a::l)::(inserer_tete a l1);;
inserer_tete : 'a -> 'a list list -> 'a list list = <fun>

#let rec map f = function
         [] -> []
       | x::l -> (f x)::map f l;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
#let inserer_tete a = map (function l -> a::l);;
```

```
inserer_tete : 'a -> 'a list list -> 'a list list = <fun>

#(* on ecrit un shuffle qui n'utilise qu'inserer_tete*)
let rec shuffle = fun
          l [] -> [l]
      |   [] l -> [l]
      |((a::l') as l) ((b::m') as m) ->
       (inserer_tete a (shuffle l' m))@(inserer_tete b (shuffle l m'));;
shuffle : 'a list -> 'a list -> 'a list list = <fun>

(* version plus portable : *)
let rec shuffle l1 l2 =
     match (l1,l2) with
        (_,[]) -> [l1]
   |   ([], _) -> [l2]
   |(a::reste1,b::reste2) ->
    (inserer_tete a (shuffle reste1 l2))@(inserer_tete b (shuffle l1 reste2));;

(* version avec une complexite bien meilleure *)

let rec inserer_tete a ll h =
  match ll with
    [] -> h
  | (l::ll') -> inserer_tete a ll' ( (a::l)::h ) ;;


let rec shuffleaccu = fun x y -> match (x,y) with
  (l,[]) -> [l]
| ([],l) -> [l]
| ((a::x'),(b::y')) ->
  let h1 = inserer_tete a (shuffleaccu x' y) []  in
  let h2 = inserer_tete b (shuffleaccu x y') h1 in
  h2
;;
```