TD 2

1) Factorielle :

```
let rec fact (n:int) : int =
  if ( n = 0) then 1
  else  n * fact (n-1)
```

2) Sommes :

a/ n premiers nombres entiers strictement positifs :

```
let rec sum_n (n:int) : int =
 if n = 0 then 0
 else n + sum_n(n - 1)
```

b/ somme de n premiers nombres pairs positifs :

```
let rec sum_p (n:int) : int =
 if(n > 0) then
   if (n = 1) then 2
   else ((2*n) +(sum_p(n-1)))
 else 0
```

c/ sum_f (f : int -> int) (n : int) : int qui calcule la somme (f n) + (f (n-1) + ... + f(0) :

```
let rec sum_f (f : int -> int) (n:int) : int =
 if (n = 0) then (f 0)
 else (f n) + (sum_f f (n-1));;

let sum_p2 (n:int) : int =
 let rec sum_f (f : int -> int) (n :int) :  int =
   if (n = 0) then (f 0)
   else (f n) + (sum_f f (n-1))
 in sum_f (fun n -> 2*n) n;;
```

3) Suites

a/ u : int -> int telle que (u n) donne un

```
let rec u (n:int) : int =
 if (n = 0) then 42
 else ((3*u(n-1))+4)
```

b/ sum_u qui calcule la somme des n + 1 premiers termes de la suite un.

```
let rec sum_u (n:int) : int =
 if (n = 0) then 42
 else (u n) + sum_u(n - 1
```

c/ Sans utiliser la fonction u, définir une fonction de signature loop (n:int) (t:int) : int qui calcule la somme t + v(t) + v2(t) + ... + vn(t) où v(x) = 3x+4. Indication : l'argument t est modifié à chaque itération: 3 * t + 4.

```
let sum_u2 (n:int) : int =
 let rec loop (n:int) (t:int) : int =
   if (n = 0) then t
   else (t + (loop (n-1)(3*t+4)))
 in loop n 42
```

4) Recurrence sur une intervalle

a/ (sum_inter a b) donne la somme des entiers compris dans l'intervalle [a,b]

```
let rec sum_inter (a:int) (b:int) : int =
 if a > b then 0
 else b + sum_inter a (b-1)
```

b/ sum1_inter (k:int) (a:int) (b:int) : int qui donne la somme (k+a) + (k+a+1) + ... + (k+b).

```
let rec sum1_inter (k:int) (a:int) (b:int) : int =
 if a > b then 0
 else
 if a = b then k + a
```

```
else (k+b) + (sum1_inter k a (b-1))
```

c/ sum2_inter (a:int) (b:int) : int qui donne la somme de tous les couples d'entiers de l'intervalle [a,b].
Par exemple, (sum2_inter 2 4) = (2+2)+(2+3)+(2+4)+(3+2)+(3+3)+(3+4)+(4+2)+(4+3)+(4+4).

```
let sum2_inter (a:int) (b:int) : int =
  let rec loop (n:int):int =
    if a > b then 0
    else if n = a then (sum1_inter n a b )
    else (sum1_inter n a b) + (loop (n - 1))
  in loop b
```

5) Nombres premiers

a/ less_divider : int -> int -> int telle que (less_divider i n) donne le plus petit diviseur de n compris entre i (inclus) et n (exclu)

```
let rec less_divider (i:int) (n:int) : int =
  if i >= n then 0
  else if n mod i = 0 then i
  else less_divider (i + 1) n
```

b/ On numérote les nombres premiers de cette manière : 2 a le numéro 0, 3 a le numéro 1, 5 a le numéro 2, 7 a le numéro 3 , etc. Déduire de la question précédente la définition de la fonction nth_prime: int -> int telle que (nth_prime n) donne le nombre premier de numéro n

```
let nth_prime (n:int):int =
  let rec loop (i:int)(j:int):int=
    if (i=0) then j
    else  loop (i-1) (next_prime(j+1))
  in loop n 2
```

TD 3 - filtrage avec les listes

1) Plus ou moins 3

a/ len_eq_3 qui donne true ssi xs contient EXACTEMENT 3 elements

```
let len_eq_3 (xs:'a list) : bool =
  match xs with
  |_::_::_::[] -> (true)
  |_ -> (false)
```

b/ len_ge_3 qui donne true ssi xs contient AU MOINS 3 elements

```
let len_ge_3 (xs:'a list) : bool =
  match xs with
  |[] -> (false)
  |_::[] -> (false)
  |_::_::[] -> (false)
  |_ -> (true)
```

c/ len_lt_3 qui donne true ssi xs contient STRICTEMENT MOINS de 3 elements

```
let len_lt_3 (xs:'a list) : bool =
  match xs with
  |[] -> (true)
  |_::[] -> (true)
  |_::_::[] -> (true)
  |_ -> (false)
```

d/ len_comp_3 qui donne :
1 si xs contient au moins 3 éléments:
0 si xs contient exactement 3 éléments;
-1 sinon

```
let len_comp_3 (xs:'a list) : int =
  if (len_eq_3 xs) then 0
  else if (len_ge_3 xs) then 1
```

```
    else -1
```

2) Premier et deuxieme

a/ snd qui donne le 2eme element de xs

```
let snd (xs:'a list) : 'a =
  match xs with
  |[] -> raise Not_found
  |_::[] -> raise Not_found
  |x1::x2::xss -> x2
```

b/ swap qui inverse les 2 premiers elements de la liste (xs si elle a moins de 2 elements)

```
let swap_hd_snd (xs:'a list) : 'a list =
  match xs with
  |[] -> xs
  |_::[] -> xs
  |x1::x2::xss -> x2::x1::xss
```

c/ hd_0 qui donne true ssi son 1er element egal à 0

```
let hd_0 (xs:int list) : bool =
  match xs with
  |0::[] -> (true)
  |0::xss -> (true)
  |_ -> (false)
```

d/ eq_hd qui donne true ssi le 1er element xs est egal à x

```
let eq_hd (x0:'a) (xs:'a list) : bool =
  match xs with
  |y::xss -> if y=x0 then (true) else (false)
  |_ -> (false)
```

3) Liste de couples, liste de listes

a/ hd_fst qui donne la première valeur du premier élément de xs. Par exemple : (hd_fst [('a',0); ('b',1); ('c',5)]) donne a.

```
let hd_fst (xs:('a*'b) list) : 'a =
  match xs with
  |[] -> raise (Not_found)
  |(a,b)::xss -> a
```

b/ swap_hd_fst (xs:('a*'a) list) : ('a*'a) list qui donne la lsite obtenue en inversant l'ordre des valeurs dans le premier élément de xs.

```
let swap_hd_fst (xs:('a*'a) list) : ('a*'a) list =
  match xs with
  |[] -> []
  |(a,b)::xss -> (b,a)::xss
```

c/ hd_hd (xs:('a list) list) : 'a qui donne le premier élément du premier élément de xs. Par exemple (hd_hd [[1;2;3]; [4]])donne 1.

```
let hd_hd (xs:('a list) list) : 'a =
  match xs with
  |[] -> raise (Not_found)
  |x1::xss -> match x1 with
    |[] -> raise (Not_found)
    |y::xss -> y
```

d/ rem_hd_hd (xs:('a list) list) : ('a list) list qui donne la liste obtenue en supprimant le premier élément du premier élément de xs. La fonction rem_hd_hd renvoie xs si la suppression n'est pas possible.

```
let rem_hd_hd (xs:('a list) list) : (int list) list =
  match xs with
  |[] -> xs
  |x1::xss -> match x1 with
    |[] -> xs
```

```
|y::x1 -> x1::xss
```

TD 4 : listes et recurrence

1) Construction de listes par recurrence sur un entier

a/ repaeat qui donne la liste contenant n fois x

```
let rec repeat (n:int) (x:'a) : 'a list  =
 if (n <= 0) then []
 else x :: repeat (n-1) (x)
```

b/range_i qui donne la liste [i, i+1, ...j]

```
let rec range_i (i:int) (j:int) : (int list) =
 if (i > j) then []
 else i :: range_i (i+1) j
```

c/range_n qui donne la liste [x; x+1; ..:x+n-1]

```
let  range_n (x:int) (n:int) : (int list) =
 let rec range_n_term (a : int) (b : int) =
 if ( a > b) then [] else
 a :: range_n_term (a + 1) b
 in
 if (n <= 0) then []
 else range_n_term x (x + n -1)
```

2) Manipulation de liste par recurrence sur les listes

a/intercale1 qui intercale z entre les éléments de xs. Le premier et le dernier éléments de (intercale1 z xs) sont le premier et le dernier éléments de xs.

```
let rec intercale1 (z:'a) (xs:'a list) : 'a list =
 match xs with
 | [] -> []
 | x :: [] -> xs
 |x::xss -> x :: z :: (intercale1 z xss)
```

b/ intercale2 qui intercale z entre les éléments de xs, mais cette fois, le premier et le dernier éléments de (intercale2 z xs) sont z.

```
let rec intercale2 (z:'a) (xs:'a list) : 'a list =
 match xs with
 | [] -> z :: []
 |x :: xss -> z :: x :: (intercale2 z xss)
```

c/begaie qui donne une liste de taille double de xs dupliquant chacun de ses éléments.

```
let rec begaie (xs:'a list) : ('a list) =
 match xs with
 | x :: xss -> x :: x :: (begaie xss)
 | _ -> xs
```

d/oublie1 qui donne la liste obtenue en oubliant un élément sur deux de xs. Le premier élément de (oublie1 xs) est le premier élément de xs.

```
let rec oublie1 (xs:'a list) : ('a list) =
 match xs with
 | x :: y :: xss -> x :: (oublie1 xss)
 | _ -> xs
```

d/ oublie2 qui donne la liste obtenue en oubliant un élément sur deux de xs, mais cette fois, le premier élément de (oublie2 xs)est le deuxième élément de xs

```
let rec oublie2 (xs:'a list) : ('a list) =
 match xs with
 | x :: y :: xss -> y :: (oublie2 xss)
 |_ -> []
```

3) Construction de listes par schema d'application

a/inverse_f Schématiquement, (inverse_f [x1; ..; xn]) donne la liste [1.0 /. x1; ..; 1.0 /. xn]

```
let rec inverse_f (xs:float list) : float list =
```

```
  match xs with
  | [] -> []
  | x :: xss -> 1.0 /. x :: (inverse_f xss)
```

b/ inberse_i Schématiquement, on a aussi que (inverse_i [x1; ..; xn]) donne la liste [1 / x1; ..; 1 / xn]

```
let rec inverse_i (ns:int list) : float list =
  match ns with
  | [] -> []
  | x :: xss -> (1. /. float_of_int x) :: inverse_i xss
```

c/ecrete qui donne la liste des éléments de xs où
toutes les valeurs inférieures (strictement) à -10 ont remplacées par -10;
toutes les valeurs supérieures (strictement) à 10 ont remplacées spar 10

```
let rec ecrete (xs:int list) : int list =
  match xs with
  |[] -> []
  | x :: xss -> if (x < -10) then -10 :: ecrete xss
     else if (x > 10) then 10 :: ecrete xss
     else x :: ecrete xss
```

d/ Schématiquement (dpoints [x1; ..; xn]) vaut [(x1, a*x1 + b); ..; (xn, a*xn + b)]

```
let rec dpoints (xs:int list) (a:int) (b:int) : (int*int) list =
  match xs with
  | [] -> []
  | x :: xss -> (x, a * x + b) :: dpoints xss a b
```

e/ Schématiquement; (app_list f [x1; ..; xn]) donne une liste égale à [(f x1); ..; (f xn)].

```
let rec app_list (f:'a -> 'b) (xs:'a list) : 'b list =
  match xs with
  | [] -> []
  | x :: xss -> (f x) :: app_list f xss
```

4) Construction de sous listes par schema de filtrage (selection)

a/ list_impair qui donne la liste des valeurs imparires de ns

```
let rec list_impair (ns:int list) : int list =
  match ns with
  | [] -> []
  | x :: xss -> if (x mod 2 = 0) then list_impair xss
     else x :: list_impair xss
```

b/ liste_non_nulle qui donne la liste des chaines de caracteres non nulles de xs

c/list_interval qui donne la liste des elemesnts de ns qui sont compris entre -10 et 10

```
let rec list_interval (ns:int list) : int list =
  match ns with
  | x :: nss -> if (x >= -10 && x <= 10) then x :: list_interval nss
     else list_interval nss
  | _ -> ns
```

d/liste_non_vide qui donne la liste des listes non vides de xss

```
let rec list_non_vide (xss:('a list) list) : ('a list) list =
  match xss with
  |[] -> []
  | x :: ns -> match x with
    | [] -> list_non_vide ns
    | y :: x -> (y :: x) :: list_non_vide ns
```

e/ list_sum_tuple qui donne la liste des couples de cs dont la somme des elements est superieure ou egale à s

```
let rec list_sum_tuple (cs:(int * int) list) (s:int) : (int * int) list =
  match cs with
```

```
 | [] -> []
 | (x,y) :: xs -> if ((x+y) >= s) then (x,y) :: list_sum_tuple xs s
    else list_sum_tuple xs s
```

e/liste_non_0 qui donne la liste des listes de nss qui ne contiennent pas 0 (utilisation de list.mem)

```
let rec list_non_0 (nss:(int list) list) : ('int list) list =
  match nss with
  | [] -> []
  | x :: xs -> if (List.mem 0 x) then list_non_0 xs
     else x :: list_non_0 xs
```

5) Construction de listes par schema de reduction

a/ Schématiquement, (prod [x1; ..; xn]) donne x1 *. .. *. xn. On a également que (prod []) donne 1.0.

```
let rec prod (xs:float list) : float =
 match xs with
 | [] -> 1.0
 | x :: xs -> x *. prod xs
```

b/ sum_round qui donne la somme des valeurs arrondies par défaut des valeurs de xs (int_of_float)

```
let rec sum_round (xs:float list) : int =
 match xs with
 | [] -> 0
 | x :: xss -> (int_of_float x) + sum_round xss
```

c/ parenthese qui donne la chaine de caracteres obtenue en mettant entre parentheses et en concatenant les elements de xs. Operateur de concatenation des chaines de caracteres ^

```
let rec parenthese (xs:string list) : string =
 match xs with
 | [] -> ""
 | x :: xss -> "(" ^ x ^ ")" ^ parenthese xss
```

d/Shématiquement: (flatten [xs1; ..: xsn]) est égal à la liste xs1@ .. @xsn, où @ est l'opérateur de concaténation des listes.

```
let rec flatten (xss:('a list) list) : 'a list =
 match xss with
 | [] -> []
 | x :: y -> x @ flatten y
```

e/Schématiquement: (sum_tuple [(x1,y1); ..; (xn,yn)])donne l'entier (x1 + y1 + .. + xn + yn).

```
let rec sum_tuple (cs:(int*int) list) : int =
 match cs with
 | [] -> 0
 | (x,y) :: xs -> x + y + sum_tuple xs
```

f/Schématiquement; (reduce f [x1; ..; xn] b) donne la valeur de (f x1 .. (f xn b)..)

```
let rec reduce (f:'a -> 'b -> 'b) (xs:'a list) (b:'b) : 'b =
 match xs with
 | [] -> b
 | x :: xss -> (f x) (reduce f xss b)
```

TD 4. bis : tri de fusion

a/ merge qui donne la liste obtenue par interclassement des elements de xs et ys

```
let rec merge (xs:'a list) (ys:'a list) : 'a list =
  match xs, ys with
  | [],_ -> ys
  |_,[] -> xs
  | x :: xss, y :: yss -> if (x < y)
     then x :: merge xss ys
     else y :: merge xs yss
```

b/ Schématiquement (split [x1; x2; x3; x4; ...]) donne le couple de listes ([x1; x3; ...], [x2; x4; ...]).

```
let rec split (xs:'a list) : ('a list * 'a list) =
  match xs with
  | [] -> [],[]
  | x::[] -> x::[],[]
  | x :: y:: xss -> let (a,b) = split xss in (x :: a, y :: b)
```

c/merge_sort qui donne une liste ordonnée par < des elements de xs

```
let rec merge_sort (xs:'a list) : 'a list =
  match xs with
  |[] -> []
  |x::[] -> xs
  |x :: ys -> let (a, b) = split xs in
     merge (merge_sort a) (merge_sort b)
```

d/ merge_gen qui donne l'interclassement des elements de listes xs et ys en utilisant la comparaison cmp. On aura que x est placé avant y lorsque (cmp x y) vaut true

```
let rec merge_gen (cmp:'a -> 'a -> bool) (xs:'a list) (ys:'a list) : 'a list =
  match xs, ys with
  | xs,[] -> xs
  |[],ys -> ys
  | x :: xss, y :: yss -> if (cmp x y) = true
     then x :: merge_gen cmp xss ys
     else y :: merge_gen cmp xs yss
```

e/merge_sort_gen qui donne la liste des elements de xs triés selon l'orduit induit pas cmp

```
let rec merge_sort_gen (cmp:'a -> 'a -> bool) (xs:'a list) : 'a list =
  let rec sorted l =
    match l with
      [] -> true
    |[x] -> true
    |x :: y :: xs -> (if (not (cmp x y)) then false
                 else sorted (y::xs))
  in
  let boolean = sorted xs in
  if (boolean) then xs
  else
    let (l1,l2) = split xs in
    let l = merge_gen cmp l1 l2 in
    merge_sort_gen cmp l
```

f/ En utilisant merge_sort_gen, définir la fonction sort qui donne la liste ordonnée des couples de xsen utilisant la fonction de comparaison définie par (x1,x2) est inférieur à (y1,y2) si (x1+x2) < (y1+y2).

```
let sort (xs:(int*int) list) : (int*int) list =
  match xs with
    [] -> []
  |x1 :: [] -> x1 ::[]
  |x1::x2::reste -> let loop (t:(int*int)) (v:(int*int)) : bool =
                 let (t1,t2) = t
                 in let (v1,v2) = v
                 in t1 + t2 < v1 + v2
     in merge_sort_gen loop xs
```