Corrigé de la feuille d'exercices n°6.

1.

```
#let hd = function
 x::xs -> x
| _ -> failwith "liste vide";;
hd : 'a list -> 'a = <fun>

#let tl = function
 x::xs -> xs
| _ -> failwith "liste vide";;
tl : 'a list -> 'a list = <fun>

let rec rev = function
 x::xs -> (rev xs)@[x]
| [] -> [];;

let rec max_liste = function
 [] -> failwith "liste vide"
| [x] -> x
| x::xs -> max x (max_liste xs);;

#let rec list_length = function
  [] -> 0
| x::xs -> 1+(list_length xs);;
list_length : 'a list -> int = <fun>

#let rec dernier = function
 [] -> failwith " erreur : liste vide"
| [x] -> x
| x::xs -> dernier xs;;
dernier : 'a list -> 'a = <fun>

#let rec elimine x = function
  [] -> []
| y::ys when y=x -> elimine x ys
| y::ys -> y::(elimine x ys);;
elimine : 'a -> 'a list -> 'a list = <fun>

#let rec doublon = function
  [] -> []
| x::xs -> x::(doublon(elimine x xs));;
doublon : 'a list -> 'a list = <fun>

let rec map f = function
 [] -> []
| x::xs -> (f(x))::(map f xs);;

#let rec applique  = fun
 (f::fs) (x::xs) -> (f(x))::(applique fs xs)
| [] [] -> []
| _  _ -> failwith "les listes n'ont pas la même longueur";;
applique : ('a -> 'b) list -> 'a list -> 'b list = <fun>

#let rec liste_it = fun
  f [] -> failwith "erreur : liste vide"
| f [x] -> failwith "erreur : liste trop courte"
| f [x;y] -> f(x,y)
| f (x::xs) -> f(x,(liste_it f xs));;
liste_it : ('a * 'a -> 'a) -> 'a list -> 'a = <fun>

#let permg = function
  x::xs -> xs@[x]
| [] -> [];;
permg : 'a list -> 'a list = <fun>

#let permd l =
let rec dernier = function
  [] -> failwith "erreur : liste vide"
| [x] -> (x,[])
| x::xs -> let (d,l) = dernier xs in (d,x::l)
in
let (a,l) = dernier l in
  a::l;;

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

```
let rec produit = function
 [] -> 0
| [x] -> 1
| [x;y] -> (y-x)
| x::xs -> (list_it (mult_int) (map (fun u -> u-x) xs)) 1
                * (produit xs);;
produit : int list -> int = <fun>
```

2. La fonction `appartient` détermine si une élément x appartient à un ensemble sous forme de liste triée. La liste étant triée en ordre croissant, on peut s'arrêter dès que x est supérieur strict à la tête de la liste.

```
let rec appartient x = function
 [] -> false
| y::ys when x=y -> true
| y::ys -> if x > y then appartient x ys
                    else false;;
appartient : 'a -> 'a list -> bool = <fun>
```

La fonction `intersect` calcule l'intersection entre deux ensembles représentés par deux listes rangées en ordre croissant. On parcourt une seule fois chacune des listes.

```
#let rec intersect = fun
  [] l2 -> []
| l1 [] -> []
| (x::xs) (y::ys) when x=y -> x::(intersect xs ys)
| (x::xs as l1) (y::ys as l2) -> if x < y then (intersect xs l2)
                                          else (intersect l1 ys);;
intersect : 'a list -> 'a list -> 'a list = <fun>
#intersect [1;2;6;8;9] [2;3;4;5;6;7;9];;
- : int list = [2; 6; 9]
```

La fonction `union` calcule la réunion entre deux ensembles représentés par deux listes rangées en ordre croissant. On parcourt une seule fois chacune des listes.

```
#let rec union = fun
  [] l2 -> l2
| l1 [] -> l1
(* cas où les 2 têtes de listes sont égales *)
| (x::xs) (y::ys) when x=y -> x::(union xs ys)
| (x::xs as l1) (y::ys as l2) -> if x < y then x::(union xs l2)
                                          else y::(union l1 ys);;
union : 'a list -> 'a list -> 'a list = <fun>
#union [1;2;6;8;9] [2;3;4;5;6;7;9];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

La fonction `diff` calcule l'intersection $A \setminus B$ entre deux ensembles A et B représentés par deux listes rangées en ordre croissant. On parcourt une seule fois chacune des listes.

```
#let rec diff = fun
  [] l2 -> l2
| l1 [] -> []
| (x::xs) (y::ys) when x=y -> (diff xs ys)
| (x::xs as l1) (y::ys as l2) -> if x < y then (diff xs l2)
                                          else y::(diff l1 ys);;
diff : 'a list -> 'a list -> 'a list = <fun>
#diff [1;2;6;8;9] [2;3;4;5;6;7;9];;
- : int list = [3; 4; 5; 7]
```

La fonction `parties` génère l'ensembles des parties d'un ensemble.

```
#let parties ensemble =
let rec partie = function
 [] -> [[]]
| x::xs -> let A = partie xs in
           union (map (fun l -> x::l) A) A in
partie ensemble;;
parties : 'a list -> 'a list list = <fun>
#parties [1;2;3];;
- : int list list = [[1; 2; 3]; [1; 2]; [1; 3];
                     [1]; [2; 3]; [2]; [3]; []]
```