Annales de programmation fonctionnelle et logique 2000-2001

1 TD 1 : définitions, filtrage

1.1 Définitions

1. Définissez une variable a égale à 3*b où b vaut 7. Attention, nous ne voulons pas que b soit définie globalement !

```
# let a = let b = 7 in 3*b;;
val a : int = 21
```

2. Définissez une fonction qui à l'entier x associe x+a où a vaut localement 3.

```
# let f x = let a = 3 in x+a;;
val f : int -> int = <fun>
```

3. Définissez une fonction f qui à l'entier x associe l'entier 2*x+1.

```
# let f x = 2*x+1;;
val f : int -> int = <fun>
```

4. Définissez une fonction g qui à l'entier x associe $(2*x+1)^2$.

```
# let g x = (2*x+1)*(2*x+1);;
val g : int -> int = <fun>
```

5. Même question que précédemment, mais en ne calculant qu'une fois la valeur de 2*x+1 (et sans utiliser de fonction "carré").

```
# let g x = let a = 2*x+1 in a*a;;
val g : int -> int = <fun>
```

6. Même problème qu'à la question 4, mais en utilisant une fonction locale pour calculer le produit 2*x+1.

```
# let g x = let f y = 2*y+1 in (f x)*(f x);;
val g : int -> int = <fun>
```

7. Même problème qu'à la question précédente, mais en n'effectuant qu'un seul appel à la fonction locale.

```
# let g x = let f y = 2*y+1 in let a = f x in a*a;;
val g : int -> int = <fun>
```

```
# let g x = let a = (let f y = 2*y+1 in f x) in a*a;;
val g : int -> int = <fun>
```

1.2 Filtrage

Attention : tous les filtrages doivent être exhaustifs !

1. Nous voulons trois versions d'une fonction qui prend une paire d'entiers en argument et qui renvoie la somme des éléments de la paire :
   (a) une version simple (sans filtrage) ;
   (b) une version avec filtrage par fonction anonyme ;
   (c) une version avec filtrage explicite (et prenant en entrée un unique argument p).

```
#let sum (a,b) = a+b;;
val sum : int * int -> int = <fun>

#let sum = function
   (a,b) -> a + b;;
val sum : int * int -> int = <fun>

#let sum p = match p with
   (a,b) -> a + b;;
val sum : int * int -> int = <fun>
```

2. Écrivez une fonction qui prend en entrée une liste et qui renvoie la somme des deux premiers entiers la constituant. Proposez plusieurs solutions (avec et sans filtrage (explicite ou non)).

```
#let somme_deux (a::b::r) = a+b;;
# let somme_deux (a::b::r) = a+b;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val somme_deux : int list -> int = <fun>

#let somme_deux = function
     []    -> failwith "pas assez d'arguments"
  |  [a]   -> failwith "pas assez d'arguments"
  | a::b::r -> a+b;;
val somme_deux : int list -> int = <fun>

#let somme_deux = function
   a::b::r -> a+b
  |   _    -> failwith "pas assez d'arguments";;
val somme_deux : int list -> int = <fun>

#let somme_deux l = match l with
   a::b::r -> a+b
  |   _    -> failwith "pas assez d'arguments";;
val somme_deux : int list -> int = <fun>
```

3. Écrivez deux versions d'une fonction qui prend en entrée les coordonnées de deux points (sous forme de paires de réels) et qui renvoie le carré de la longueur du segment correspondant : une version simple et une avec un filtrage explicite (la fonction prend alors en entrée deux arguments `a` et `b`).

```
#let longueur (a,b) (c,d) = (a-.c)*.(a-.c) +. (b-.d)*.(b-.d);;
val longueur : float * float -> float * float -> float = <fun>

#let longueur a b =
 match (a,b) with ((x,y),(z,t)) -> (x-.z)*.(x-.z) +. (y-.t)*.(y-.t);;
val longueur : float * float -> float * float -> float = <fun>
```

4. Écrivez une fonction qui prend en entrée une liste de paires et qui renvoie la somme des éléments de son premier élément (sic).

```
#let liste_de_paires = function
   (a,b)::_ -> a+b
  |   _      -> failwith "liste vide";;
val liste_de_paires : (int * int) list -> int = <fun>
```

Exemple d'utilisation :

```
#liste_de_paires [(3,4);(5,6);(7,8)];;
- : int = 7
```

5. Écrivez une fonction qui prend en argument une paire de listes et qui renvoie la somme du premier élément de la première liste et du premier élément de la deuxième.

```
#let paire_de_listes = function
   (a::_,b::_) -> a+b
  |      _      -> failwith "une des deux listes est vide";;
val paire_de_listes : int list * int list -> int = <fun>
```

6. Même question que précédemment, mais sans utiliser le symbole « _ ».

```
#let paires_de_listes = function
     ([],[])   -> failwith "les deux listes sont vides"
  |  ([],t::r) -> failwith "la première liste est vide"
  | (t::r,[])  -> failwith "la deuxième liste est vide"
  | (a::b,c::d) -> a+c;;
val paires_de_listes : int list * int list -> int = <fun>
```

7. Écrivez une fonction qui prend en entrée une paire et une liste et qui fait la somme des premiers éléments de ses deux arguments.

```
#let paire_et_liste p l = match (p,l) with
   ((a,_),t::_) -> a+t
  |      _      -> failwith "Le deuxième argument est une liste vide";;
val paire_et_liste : int * 'a -> int list -> int = <fun>
```

Exemple d'utilisation :

```
#paire_et_liste (1,3) [7;12;45];;
- : int = 8
```

8. Écrivez une fonction qui prend en entrée une liste de trois entiers, notée `[a ;b ;c]` et qui renvoie la liste `[c ;b ;a ;b ;c]`. Cette fonction devra utiliser un synonyme.

```
#let liste_avec_synonyme = function
   [a;b;c] as l -> c::b::l
  |     _       -> failwith "la liste doit avoir exactement trois arguments";;
val liste_avec_synonyme : 'a list -> 'a list = <fun>
```

2 TD 2 : récursivité

Expressions à évaluer

```
#let x = 1;;

#let y = 4;;

#let x = 3 and y = x + 1 in
   let x = x + y and y = x - y
     in (x,y);;

#let x = 3
   in let  y = x + 1
     in let x = x + y
       in let y = x - y
         in (x,y);;
```

```
#let x = 1;;
val x : int = 1

#let y = 4;;
val y : int = 4

#let x = 3 and y = x + 1 in
   let x = x + y and y = x - y
     in (x,y);;
- : int * int = 5, 1

#let x = 3
   in let  y = x + 1
     in let x = x + y
       in let y = x - y
         in (x,y);;
- : int * int = 7, 3
```

Fonctions élémentaires sur les listes

Écrire une fonction :

1. Qui rend `true` si et seulement si une liste n'a qu'un seul élément ;

```
#let un_seul_élément = function
    []      -> false
  | t::r::l -> false
  | _       -> true;;
val un_seul_élément : 'a list -> bool = <fun>

#let un_seul_élément = function
    [a] -> true
```

```
  |  _  -> true;;
val un_seul_élément : 'a list -> bool = <fun>
```

2. Qui renvoie le second élément d'une liste d'au moins deux éléments ;

```
#let second_élément = function
     []       -> failwith "second_élément"
   | t::r::l -> r
   | _        -> failwith "second_élément";;
val second_élément : 'a list -> 'a = <fun>
```

```
#let second_élément = function
   | t::r::l -> r
   | _        -> failwith "second_élément";;
val second_élément : 'a list -> 'a = <fun>
```

3. Qui calcule la longueur d'une liste ;

```
#let rec longueur = function
      []   -> 0
    | t::r -> 1+ longueur r;;
val longueur : 'a list -> int = <fun>
```

4. Qui calcule la somme des éléments d'une liste de nombres ;

```
#let rec somme = function
      []   -> 0
    | t::r -> t + somme r;;
val somme : int list -> int = <fun>
```

5. Qui rend le dernier élément d'une liste non vide ;

```
#let rec dernier = function
      []    -> failwith "dernier"
    | t::[] -> t
    | _::r  -> dernier r;;
val dernier : 'a list -> 'a = <fun>
```

6. Qui prend une liste et un élément en arguments et renvoie `true` si et seulement si l'élément appartient à la liste ;

```
#let rec appartient_a x = function
      []   -> false
    | t::r -> t=x or appartient_a x r;;
val appartient_a : 'a -> 'a list -> bool = <fun>
```

7. Qui teste l'égalité de deux listes ;

```
#let rec égales l1 l2 = match (l1,l2) with
          [],[]      -> true
    | t1::r1,t2::r2 -> t1=t2 & égales r1 r2
    |      _,_       -> false;;
val égales : 'a list -> 'a list -> bool = <fun>
```

```
#let rec égales l1 l2 = match (l1,l2) with
          [],[]      -> true
    | t1::r1,t2::r2 -> t1=t2 & égales r1 r2
    |       _       -> false;;
val égales : 'a list -> 'a list -> bool = <fun>
```

8. Qui concatène deux listes ;

```
#let rec concatene l1 l2 =
   match l1 with
      []   -> l2
    | t::r -> t :: concatene r l2 ;;
val concatene : 'a list -> 'a list -> 'a list = <fun>
```

9. Qui renverse une liste (écrire une fonction avec et une fonction sans accumulateur) ;

```
# let rec renverse = function
      []    -> []
    | t::r -> concatene (renverse r)  [t] ;;
val renverse : 'a list -> 'a list = <fun>
```

```
# let renverse l =
    let rec renverse_accu accu = function
        []    -> accu
      | t::r -> renverse_accu (t::accu) r
    in renverse_accu [] l;;
val renverse : 'a list -> 'a list = <fun>
```

10. Qui insère un élément dans une liste triée (la liste obtenue doit bien évidemment être triée) ;

```
# let rec insere x = function
      []        -> [x]
   | t::r as l -> if x<t then x::l
                         else t:: insere x r;;
val insere : 'a -> 'a list -> 'a list = <fun>
```

11. Qui trie par insertion une liste.

```
# let rec tri = function
      []   -> []
   | t::r -> insere t (tri r);;
val tri : 'a list -> 'a list = <fun>
```

```
# tri [6;2;8;4;9];;
- : int list = [2; 4; 6; 8; 9]
```

3 TD 3 : récursivité

Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$\mathrm{Fib}(n) = \begin{cases} 1 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \text{ sinon.} \end{cases}$$

1. Écrire une fonction récursive qui calcule $\mathrm{Fib}(n)$.

```
# let rec fib_filtre =
  function
     0 -> 1
   | 1 -> 1
   | n -> (fib_filtre (n-1)) + (fib_filtre (n-2));;
val fib_filtre : int -> int = <fun>
```

2. Écrire une fonction récursive qui calcule, pour $n > 0$, $(\mathrm{Fib}(n), \mathrm{Fib}(n-1))$. L'utiliser pour écrire une fonction calculant $\mathrm{Fib}(n)$.

```
# let rec fib_paire =
  function
     0 -> failwith "L'argument est nul."
   | 1 -> (1,1)
   | n -> let (x,y)= fib_paire (n-1) in(x+y,x);;
val fib_paire : int -> int * int = <fun>
```

```
# let fib = function
     0 -> 1
   | n -> let (x,y)= fib_paire n in x;;
val fib : int -> int = <fun>
```

Calcul de (cos(nx), sin(nx))

Écrire une fonction qui prend en entrée un entier `n` et une paire de valeurs réelles qui sont en fait les valeurs du cosinus et du sinus d'un certain angle `x`, et qui renvoie la paire `(cos(nx), sin(nx))`. Autrement dit, le deuxième argument de la fonction est une paire `(a,b)` telle que `a = cos x` et `b = sin x` . Le schéma de calcul doit bien évidemment être récursif.

On pourra se servir des formules de trigonométrie suivantes :

```
cos(nx) = cos((n-1)x) cos(x) - sin((n-1)x) sin(x)
sin(nx) = sin((n-1)x) cos(x) + cos((n-1)x) sin(x)
```

```
#let trigo (a, b) n =
   let rec trigo_aux = function
       0 -> (1.,0.)
     | 1 -> (a, b)
     | m -> let (c,d) = trigo_aux (m-1)
              in (a *. c -. b *. d, a *. d +. b *. c)
   in trigo_aux n;;
val trigo : float * float -> int -> float * float = <fun>
```

Tri par recherche du maximum

1. Écrire une fonction qui retourne la valeur du maximum d'une liste.

```
#let rec maximum = function
     []   -> failwith "Liste vide"
   | [a]  -> a
   | t::r -> max t (maximum r);;
val maximum : 'a list -> 'a = <fun>
```

2. Écrire une fonction qui prend une liste en entrée et retourne la liste privée de son élément maximal.
   (a) Écrire une version de cette fonction qui recherche d'abord l'élément maximum, puis qui l'enlève de la liste.

```
#maximum [3;2;4];;
- : int = 4

#let rec ote x = function
   t::r -> if (t=x) then r
                    else t::(ote x r)
  |  []  -> [];;
val ote : 'a -> 'a list -> 'a list = <fun>

#let sansmax l = ote (maximum l) l;;
val sansmax : 'a list -> 'a list = <fun>

#let sansmax l =
   let rec ote x = function
     t::r -> if (t=x) then r
                      else t::(ote x r)
    |  []  -> []
   in ote (maximum l) l;;
val sansmax : 'a list -> 'a list = <fun>

#sansmax [2;7;4;6];;
- : int list = [2; 4; 6]
```

   (b) Écrire une version de cette fonction qui calcule simultanément le maximum et la liste demandée.

```
#let sansmax =
   let rec aux x = function
        []  -> []
     | t::r -> if (t>x) then (x::(aux t r))
```

```
                        else t::(aux x r)
    in function    []  -> []
                 | t::r -> aux t r;;
val sansmax : 'a list -> 'a list = <fun>

# sansmax [2;7;4;6];;
- : int list = [2; 4; 6]
```

3. Si les paires ont été vues en cours, écrire une fonction qui prend en entrée une liste et renvoie une paire composée de l'élément maximum et de la liste des autres éléments.

```
# let maxpaire =
    let rec aux x accu = function
        []  -> (x,accu)
     | t::r -> if (t>x) then aux t (x::accu) r
                         else aux x (t::accu) r
    in function   []  -> failwith "Liste vide"
               | t::r -> aux t [] r;;
val maxpaire : 'a list -> 'a * 'a list = <fun>

# maxpaire [2;7;6;4];;
- : int * int list = 7, [4; 6; 2]
```

4. Écrire une fonction de tri de liste par recherches successives de maximum (on utilisera bien évidemment certaines des fonctions précédemment obtenues).

   (a) Écrire une fonction qui trie dans l'ordre décroissant.

```
# let rec tri = function
      [] -> []
   |  l -> let x = maximum l and liste = sansmax l
               in x::(tri liste);;
val tri : 'a list -> 'a list = <fun>

# tri [2;7;6;4];;
- : int list = [7; 6; 4; 2]

# let rec tri = function
      [] -> []
   |  l -> let (x,liste) = maxpaire l
               in x::(tri liste);;
val tri : 'a list -> 'a list = <fun>

# tri [2;7;6;4];;
- : int list = [7; 6; 4; 2]
```

   (b) Écrire une fonction qui trie dans l'ordre croissant (sans utiliser de fonctions de concaténations de listes).

```
# let tri l =
    let rec aux accu = function
        []  -> accu
     |  _ as l -> let x = maximum l and liste = sansmax l
                    in aux (x::accu) liste
    in aux [] l;;
val tri : 'a list -> 'a list = <fun>

# tri  [2;7;6;4];;
- : int list = [2; 4; 6; 7]

# let tri l1 =
    let rec aux accu l2 = match l2 with
        [] -> accu
     |  _ -> let (x,liste) = maxpaire l2
              in aux (x::accu) liste
    in aux [] l1;;
```

```
val tri : 'a list -> 'a list = <fun>

#tri [2;7;6;4];;
- : int list = [2; 4; 6; 7]

#let tri =
   let rec aux accu = function
        []   -> accu
    | _ as l -> let (x,liste) = maxpaire l
                in aux (x::accu) liste
   in aux [];;
val tri : '_a list -> '_a list = <fun>

#tri [2;7;6;4];;
- : int list = [2; 4; 6; 7]
```

4 TD 4 : typage et polymorphisme

Expressions à typer

Donnée :

```
#print_int;;
- : int -> unit = <fun>
#let f x y = x;;

#let f x y = y;;

#let f x y = if true then x else y;;

#let f x y = if false then x else y;;

#let f x y = if x=y then 0 else 1;;

#let f x y = if x=y then x else 1;;

#let f x y = if x=y then x+y else x-y;;

#let f x y = if x then x+y else x-y;;

#let f x y = if x then print_int y;;

#let f g x = g(g x);;

#let rec f a b c =
   match a with
       []   -> [b c]
    | t::r  -> (b t)::(f r b c);;

#let rec f = function
     []  -> []
  | t::r -> f r;;
#let f x y = x;;
val f : 'a -> 'b -> 'a = <fun>

#let f x y = y;;
val f : 'a -> 'b -> 'b = <fun>

#let f x y = if true then x else y;;
val f : 'a -> 'a -> 'a = <fun>

#let f x y = if false then x else y;;
val f : 'a -> 'a -> 'a = <fun>

#let f x y = if x=y then 0 else 1;;
val f : 'a -> 'a -> int = <fun>

#let f x y = if x=y then x else 1;;
```

```
val f : int -> int -> int = <fun>

#let f x y = if x=y then x+y else x-y;;
val f : int -> int -> int = <fun>

#let f x y = if x then x+y else x-y;;
# let f x y = if x then x+y else x-y;;
This expression has type bool but is here used with type int

#let f x y = if x then print_int y;;
val f : bool -> int -> unit = <fun>

#let f g x = g(g x);;
val f : ('a -> 'a) -> 'a -> 'a = <fun>

#let rec f a b c =
   match a with
       []   -> [b c]
     | t::r  -> (b t)::(f r b c);;
val f : 'a list -> ('a -> 'b) -> 'a -> 'b list = <fun>

#let rec f = function
      []  -> []
  | t::r -> f r;;
val f : 'a list -> 'b list = <fun>
```

Addition polymorphe des éléments d'une liste

1. Écrivez une fonction récursive qui calcule la somme des éléments d'une liste d'entiers, en supposant que la liste est de taille supérieure ou égale à un.

```
#let rec somme = function
     [a] -> a
   | t::r -> t + (somme r)
   |  _   -> failwith "Liste vide interdite";;
val somme : int list -> int = <fun>
```

2. Écrivez une fonction récursive qui calcule la somme des éléments d'une liste de taille supérieure ou égale à un, quel que soit le type des éléments de cette liste.

```
#let rec somme add = function
     [a] -> a
   | t::r -> add t (somme add r)
   |  _   -> failwith "Liste vide interdite";;
val somme : ('a -> 'a -> 'a) -> 'a list -> 'a = <fun>
```

3. Typez la fonction précédente.
4. Écrivez une fonction récursive qui applique récursivement une opération binaire (+, *., ^, etc.) aux éléments d'une liste de taille supérieure ou égale à un, quel que soit le type des éléments de cette liste. La solution de la question 2 répond à la présente question.
5. Même question que précédemment mais en ne supposant plus que la liste soit forcément non vide.

```
#let rec somme add e = function
     [] -> e
   | t::r -> add t (somme add e r);;
val somme : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

6. Typez la fonction précédente.

5 TD 5 : fonctionnelles sur listes

Polynômes creux

Dans ce TD nous allons manipuler des monômes définis par le type suivant :

```
# type monôme = {coefficient : int; degré : int};;
type monôme =  coefficient : int; degré : int;
```

Un polynôme (creux) sera alors constitué d'une liste de monômes, cette liste n'étant pas supposée être triée en fonction des degrés des monômes.

Affichage

1. Écrivez une fonction d'affichage d'un monôme.

```
# let affiche_monôme m =
   print_string
     ((string_of_int m.coefficient)^".X^"^(string_of_int m.degré));;
val affiche_monôme : monôme -> unit = <fun>
```

2. Écrivez, au moyen d'une fonctionnelle, une fonction qui affiche tous les monômes d'un polynôme.

```
# let affiche_polynôme p =
    let affiche_aux m =
       affiche_monôme m; print_string " "
    in List.iter affiche_aux p;;
val affiche_polynôme : monôme list -> unit = <fun>
```

Produit de deux polynômes

1. Écrivez une fonction de multiplication de deux monômes.

```
# let mul_monôme m n =
   {coefficient = m.coefficient*n.coefficient; degré = m.degré+n.degré};;
val mul_monôme : monôme -> monôme -> monôme = <fun>
```

2. Écrivez, au moyen d'une fonctionnelle, une fonction qui multiplie un polynôme par un monôme.

```
# let mul_polynôme_par_monôme m p =
   List.map (mul_monôme m) p;;
val mul_polynôme_par_monôme : monôme -> monôme list -> monôme list = <fun>
```

3. Écrivez, au moyen d'une fonctionnelle, une fonction qui multiplie deux polynômes.

```
# let mul_polynôme p q =
   let l = List.map (function x -> mul_polynôme_par_monôme x p) q
   in List.fold_left (function x -> function y -> x @ y) [] l;;
val mul_polynôme : monôme list -> monôme list -> monôme list = <fun>
```

```
# let mul_polynôme p q =
   List.fold_left
     (function x -> function y -> x @ y)
     []
     (List.map (function x -> mul_polynôme_par_monôme x p) q);;
val mul_polynôme : monôme list -> monôme list -> monôme list = <fun>
```

4. Même question que précédemment, mais en n'utilisant qu'une seule fonctionnelle... (si vous en avez utilisé deux à la question précédente).

```
# let mul_polynôme p q =
   List.fold_left
     (function x -> function y -> (mul_polynôme_par_monôme y p) @ x)
     []
     q;;
val mul_polynôme : monôme list -> monôme list -> monôme list = <fun>
```

```
# let mul_polynôme p q =
   List.fold_right
     (function x -> function y -> (mul_polynôme_par_monôme x p) @ y)
     q
     [];;
val mul_polynôme : monôme list -> monôme list -> monôme list = <fun>
```

Évaluation d'un polynôme en un point

1. Écrivez une fonction qui évalue un monôme en un point.

```
# let eval_monôme m x =
    let rec puiss = function
        0 -> 1
      | n -> x * (puiss (n-1))
    in m.coefficient * (puiss m.degré);;
val eval_monôme : monôme -> int -> int = <fun>
```

2. Écrivez, au moyen d'une fonctionnelle, une fonction qui évalue un polynôme en un point.

```
# let eval_polynôme p x =
    List.fold_right
      (function y -> function z -> z + (eval_monôme y x))
      p
      0;;
val eval_polynôme : monôme list -> int -> int = <fun>
```

```
# let eval_polynôme p x =
    List.fold_left
      (function y -> function z -> y + (eval_monôme z x))
      0
      p;;
val eval_polynôme : monôme list -> int -> int = <fun>
```

6 TD 6 : exceptions

Listes

1. Écrivez une fonction qui recherche si un élément appartient à une liste.

```
# let rec appartient x = function
    []   -> false
  | t::r -> (t = x) or (appartient x r);;
val appartient : 'a -> 'a list -> bool = <fun>
```

2. Optimisez cette fonction par l'usage d'exceptions.

```
# exception Trouvé;;
exception Trouvé
```

```
# let appartient_opt x l =
    let rec appartient_aux = function
       []  -> false
    | t::r -> if (t = x) then raise Trouvé
                         else (appartient_aux r)
    in try appartient_aux l with Trouvé -> true;;
val appartient_opt : 'a -> 'a list -> bool = <fun>
```

3. Écrivez une fonction booléenne récursive qui recherche si un élément appartient à une liste de listes.

```
# let rec appartient_liste x = function
    []   -> false
  | t::r -> (appartient x t) or (appartient_liste x r);;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

4. Optimisez cette fonction par l'usage d'exceptions.

```
# let appartient_liste x l =
    let rec appartient_aux = function
        []   -> false
      | t::r -> if t=x then raise Trouvé
                       else appartient_aux r
    in
```

```
      let rec appartient_liste_aux = function
           []  -> false
         | t::r -> (appartient_aux t) or (appartient_liste_aux r)
      in try appartient_liste_aux l with Trouvé -> true;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

5. Écrivez, au moyen de fonctionnelles, une fonction booléenne qui recherche si un élément appartient à une liste de listes.

```
# let appartient_liste a l =
    List.fold_left
       (function x-> function y-> (appartient a y) or x)
       false
       l;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

6. Optimisez cette fonction par l'usage d'exceptions.

```
# let appartient_liste a l =
   let rec appartient_aux = function
       []   -> false
     | t::r -> if t=a then raise Trouvé
                      else appartient_aux r
   in try List.fold_left
            (function x-> function y-> (appartient a y) or x)
            false
            l
      with Trouvé -> true;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

7. Écrivez une fonction récursive qui recherche si un élément appartient à une liste de listes et qui renvoie la liste contenant l'élément.

```
# let rec appartient_liste x = function
   []   -> []
  | t::r -> if (appartient x t) then t else (appartient_liste x r);;
val appartient_liste : 'a -> 'a list list -> 'a list = <fun>
```

8. Optimisez cette fonction par l'usage d'exceptions (on supposera ici que l'on traite des listes de listes d'entiers).

```
# exception Liste of int list;;
exception Liste of int list

# let appartient_liste x l =
   let rec appartient_aux = function
       []   -> false
     | t::r -> if t=x then raise Trouvé
                      else appartient_aux r
   in
      let rec appartient_liste_aux = function
           []  -> []
         | t::r -> if (try (appartient_aux t) with Trouvé -> true)
                   then raise (Liste t)
                   else (appartient_liste_aux r)
      in try appartient_liste_aux l with Liste t -> t;;
val appartient_liste : int -> int list list -> int list =
<fun>
```

9. Écrivez, au moyen de fonctionnelles, une fonction qui recherche si un élément appartient à une liste de listes et qui renvoie la liste contenant l'élément.

```
# let rec appartient_liste a l =
    List.fold_left
       (function x -> function y -> if (appartient a y) then y else x)
       []
```

```
      l;;
val appartient_liste : 'a -> 'a list list -> 'a list = <fun>
```

10. Optimisez cette fonction par l'usage d'exceptions (on supposera ici que l'on traite des listes de listes d'entiers).

```
# let rec appartient_liste a l =
    let rec appartient_aux = function
        []   -> false
      | t::r -> if t=a then raise Trouvé
                       else appartient_aux r
    in try List.fold_left
                (function x -> function y ->
                  if try (appartient a y) with Trouvé -> true
                  then raise (Liste y)
                  else x)
                []
                l
        with Liste z -> z;;
val appartient_liste : int -> int list list -> int list
= <fun>
```

Arbres binaires

Un arbre binaire est soit une feuille (un entier), soit un nœud interne (composé d'un entier, la clef, et de deux sous-arbres). Nous définissons un arbre binaire par le type suivant :

```
# type arbre = Feuille of int | Noeud of (int * arbre  * arbre);;
type arbre = Feuille of int | Noeud of (int * arbre * arbre)
```

1. Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié.

```
# let rec appartient n= function
         Feuille f   -> f = n
   | Noeud (c, fg, fd) -> c = n or
                            (appartient n fg) or
                            (appartient n fd);;
val appartient : int -> arbre -> bool = <fun>
```

2. Écrivez une fonction `appartient_opt`, nouvelle version de `appartient` optimisée par l'usage d'exceptions.

```
# exception Trouvé;;
exception Trouvé
```

```
# let appartient_opt n tree =
     let rec appartient_aux = function
        Feuille f   -> if f = n then raise Trouvé else false
      | Noeud (c, fg, fd) -> if c = n
                             then raise Trouvé
                             else (appartient_aux fg) or
                                  (appartient_aux fd)
     in try appartient_aux tree with Trouvé -> true;;
val appartient_opt : int -> arbre -> bool = <fun>
```

7 TD 7 : logique

Analyse de vérité

Au moyen d'une analyse de vérité, prouver les égalités suivantes :

1. $\neg(a \vee b) = (\neg a) \wedge (\neg b)$.

$a=0$	$a=1$
$\neg(0 \vee b) \leftrightarrow (\neg 0) \wedge (\neg b)$	$\neg(1 \vee b) \leftrightarrow (\neg 1) \wedge (\neg b)$
$\neg b \leftrightarrow 1 \wedge (\neg b)$	$\neg(1) \leftrightarrow 0 \wedge (\neg b)$
$\neg b \leftrightarrow \neg b$	$0 \leftrightarrow 0$
1	1

2. $a \rightarrow b = (\neg b) \rightarrow (\neg a)$ (c'est la formule du raisonnement par contraposée).

$a=0$		$a=1$	
$(0 \rightarrow b) \leftrightarrow ((\neg b) \rightarrow (\neg 0))$		$(1 \rightarrow b) \leftrightarrow ((\neg b) \rightarrow (\neg 1))$	
$(0 \rightarrow b) \leftrightarrow ((\neg b) \rightarrow 1)$		$(1 \rightarrow b) \leftrightarrow ((\neg b) \rightarrow 0)$	
$(0 \rightarrow b) \leftrightarrow 1$		$(1 \rightarrow b) \leftrightarrow ((\neg b) \rightarrow 0)$	
$b=0$	$b=1$	$b=0$	$b=1$
$(0 \rightarrow 0) \leftrightarrow 1$	$(0 \rightarrow 1) \leftrightarrow 1$	$(1 \rightarrow 0) \leftrightarrow ((\neg 0) \rightarrow 0)$	$(1 \rightarrow 1) \leftrightarrow ((\neg 1) \rightarrow 0)$
$1 \leftrightarrow 1$	$1 \leftrightarrow 1$	$0 \leftrightarrow (1 \rightarrow 0)$	$1 \leftrightarrow (0 \rightarrow 0)$
1	1	$0 \leftrightarrow 0$	$1 \leftrightarrow 1$
1	1	1	1

3. $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$.

$a=0$		$a=1$
$((0 \wedge b) \vee c) \leftrightarrow ((0 \vee c) \wedge (b \vee c))$		$((1 \wedge b) \vee c) \leftrightarrow ((1 \vee c) \wedge (b \vee c))$
$(0 \vee c) \leftrightarrow (c \wedge (b \vee c))$		$(b \vee c) \leftrightarrow (1 \wedge (b \vee c))$
$c \leftrightarrow (c \wedge (b \vee c))$		$(b \vee c) \leftrightarrow (b \vee c)$
$c \leftrightarrow (c \wedge (b \vee c))$		1
$b=0$	$b=1$	
$c \leftrightarrow (c \wedge (0 \vee c))$	$c \leftrightarrow (c \wedge (1 \vee c))$	
$c \leftrightarrow (c \wedge c)$	$c \leftrightarrow (c \wedge 1)$	
$c \leftrightarrow c$	$c \leftrightarrow c$	
1	1	

4. $(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$.

$a=0$	$a=1$	
$((0 \vee b) \wedge c) \leftrightarrow ((0 \wedge c) \vee (b \wedge c))$	$((1 \vee b) \wedge c) \leftrightarrow ((1 \wedge c) \vee (b \wedge c))$	
$(b \wedge c) \leftrightarrow (0 \vee (b \wedge c))$	$(1 \wedge c) \leftrightarrow (c \vee (b \wedge c))$	
$(b \wedge c) \leftrightarrow (b \wedge c)$	$c \leftrightarrow (c \vee (b \wedge c))$	
1	$c \leftrightarrow (c \vee (b \wedge c))$	
	$b=0$	$b=1$
	$c \leftrightarrow (c \vee (0 \wedge c))$	$c \leftrightarrow (c \vee (1 \wedge c))$
	$c \leftrightarrow (c \vee 0)$	$c \leftrightarrow (c \vee c)$
	$c \leftrightarrow c$	$c \leftrightarrow c$
	1	1

Remarque : prouvez l'égalité $A = B$ est équivalent à montrer que $A \leftrightarrow B$ est une tautologie.

Mise sous Forme Normale Conjonctive

Mettre sous Forme Normale Conjonctive les formules suivantes (on pourra se servir des formules démontrées à la section précédente) :

$$((p \rightarrow (q \vee r)) \wedge ((q \vee s) \rightarrow (t \wedge u))) \wedge \neg t$$

$$[p \rightarrow (q \vee r)] \wedge [(q \vee s) \rightarrow (t \wedge u)] \wedge [\neg t]$$

$p \rightarrow (q \vee r) = (q \vee r) \vee \neg p = q \vee r \vee \neg p$

$(q \vee s) \rightarrow (t \wedge u) = (t \wedge u) \vee \neg(q \vee s) = (t \wedge u) \vee (\neg q \wedge \neg s) = (t \vee (\neg q \wedge \neg s)) \wedge (u \vee (\neg q \wedge \neg s)) = (t \vee \neg q) \wedge (t \vee \neg s) \wedge (u \vee \neg q) \wedge (u \vee \neg s)$

D'où :

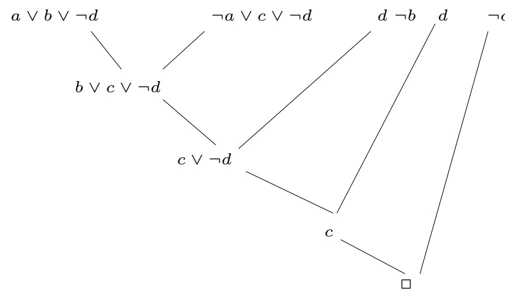
$$((p \rightarrow (q \vee r)) \wedge ((q \vee s) \rightarrow (t \wedge u))) \wedge \neg t = (q \vee r \vee \neg p) \wedge (t \vee \neg q) \wedge (t \vee \neg s) \wedge (u \vee \neg q) \wedge (u \vee \neg s) \wedge \neg t$$

$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t = (q \vee r \vee \neg p) \wedge \neg q \wedge (t \vee \neg s) \wedge (u \vee \neg q) \wedge (u \vee \neg s) \wedge \neg t$$
$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t = (q \vee r \vee \neg p) \wedge \neg q \wedge \neg s \wedge (u \vee \neg q) \wedge (u \vee \neg s) \wedge \neg t$$
$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t = (q \vee r \vee \neg p) \wedge \neg q \wedge \neg s \wedge (u \vee \neg s) \wedge \neg t$$
$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t = (q \vee r \vee \neg p) \wedge \neg q \wedge \neg s \wedge \neg t$$
$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t = (r \vee \neg p) \wedge \neg q \wedge \neg s \wedge \neg t$$

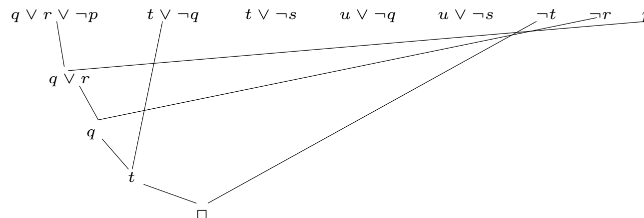
Méthode de résolution

Montrer par réfutation que :

1.
$$((a \vee b \vee \neg d) \wedge (\neg a \vee c \vee \neg d) \wedge (\neg b) \wedge d) \to c$$



2.
$$(((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t) \to (p \to r)$$



Transcription d'un énoncé

Une association est régie par le règlement intérieur suivant :

1. Les membres de la direction financière doivent être choisis parmi ceux de la direction générale. $f \to g$
2. Nul ne peut être à la fois membre de la direction générale et de la direction de la bibliothèque s'il n'est membre de la direction financière. $(g \wedge b) \to f$
3. Aucun membre de la direction de la bibiothèque ne peut être membre de la direction financière. $b \to \neg f$

On désigne par les lettre f, g et b les proposition atomiques « être membre de la direction financière », « être membre de la direction générale »,« être membre de la direction de la bibiothèque ».

Écrire sous forme décomposée l'ensemble des trois articles du règlement.

Montrer, par une analyse de vérité, que cet ensemble est sémantiquement équivalent à l'ensemble de deux propositions suivant : $\{f \to g,\ g \to \neg b\}$. Rédiger un nouveau règlement.

On veut établir la formule : $((f \to g) \wedge ((g \wedge b) \to f) \wedge (b \to \neg f)) \leftrightarrow ((f \to g) \wedge (g \to \neg b))$.

Nouveau règlement :

1. Les membres de la direction financière doivent être choisis parmi ceux de la direction générale.
2. Aucun membre de la direction générale ne peut être membre de la direction de la bibliothèque.

$b = 0$
$((f \to g) \wedge ((g \wedge 0) \to f) \wedge (0 \to \neg f)) \leftrightarrow ((f \to g) \wedge (g \to \neg 0))$
$((f \to g) \wedge (0 \to f) \wedge 1) \leftrightarrow ((f \to g) \wedge (g \to 1))$
$((f \to g) \wedge 1 \wedge 1) \leftrightarrow ((f \to g) \wedge 1)$
$(f \to g) \leftrightarrow (f \to g)$
1

<table>
<tr><th colspan="2">$b = 1$</th></tr>
<tr><td colspan="2">$((f \to g) \wedge ((g \wedge 1) \to f) \wedge (1 \to \neg f)) \leftrightarrow ((f \to g) \wedge (g \to \neg 1))$
$((f \to g) \wedge (g \to f) \wedge \neg f) \leftrightarrow ((f \to g) \wedge (g \to 0))$
$((f \to g) \wedge (g \to f) \wedge \neg f) \leftrightarrow ((f \to g) \wedge \neg g)$
$((f \to g) \wedge (g \to f) \wedge \neg f) \leftrightarrow ((f \to g) \wedge \neg g)$
$((f \to g) \wedge (g \to f) \wedge \neg f) \leftrightarrow ((f \to g) \wedge \neg g)$</td></tr>
<tr><th>$f = 0$</th><th>$f = 1$</th></tr>
<tr><td>$((0 \to g) \wedge (g \to 0) \wedge \neg 0) \leftrightarrow ((0 \to g) \wedge \neg g)$
$(1 \wedge \neg g \wedge 1) \leftrightarrow (1 \wedge \neg g)$
$\neg g \leftrightarrow \neg g$
1</td><td>$((1 \to g) \wedge (g \to 1) \wedge \neg 1) \leftrightarrow ((1 \to g) \wedge \neg g)$
$(g \wedge 1 \wedge \neg 1) \leftrightarrow (g \wedge \neg g)$
$(g \wedge 0) \leftrightarrow 0$
$0 \leftrightarrow 0$</td></tr>
</table>

8 TD 8 : premiers programmes Prolog

1. Écrire un prédicat « hypoténuse » qui prend trois arguments X, Y et Z et qui est vrai si $Z^2 = X^2 + Y^2$ (l'égalité entre expressions arithmétiques se note « = := »).

```
hypoténuse(X,Y,Z) :- X*X + Y*Y =:= Z*Z.
```

2. Écrire un prédicat qui prend en entrée un élément et une liste et qui est vrai si l'élément appartient à la liste.

```
estelt(X,[X|Y]).
estelt(X,[Y|L]) :- estelt(X,L).
```

3. Écrire un prédicat qui est vrai si son argument est un chiffre.

```
chiffre(T) :- estelt(T,[0,1,2,3,4,5,6,7,8,9]).
```

4. Réécrire le prédicat « hypoténuse » pour qu'il puisse énumérer tous les triplets (X, Y, Z) de chiffres satisfaisants la relation $X^2 + Y^2 = Z^2$.

```
hypoténuse(X,Y,Z) :- chiffre(X), chiffre(Y), chiffre(Z), X*X + Y*Y =:= Z*Z.
```

5. Écrire un prédicat qui est vrai si tous les éléments d'une liste sont des chiffres.

```
tous_chiffres([]).
tous_chiffres([T|L]) :- estelt(T,[0,1,2,3,4,5,6,7,8,9]), tous_chiffres(L).
```

   Autre solution :

```
tous_chiffres([]).
tous_chiffres([T|L]) :- between(0,9,T), tous_chiffres(L).
```

6. Écrire un prédicat prenant un élément et une liste en argument, et qui est vrai si et seulement si l'élément n'appartient pas à la liste (le symbole différent est « =\= »).

```
paselt(X,[]).
paselt(X,[Y|L]) :- paselt(X,L), X =\= Y.
```

7. Écrire un prédicat qui est vrai si tous les éléments d'une liste sont (deux à deux) différents.

```
tous_différents([]).
tous_différents([T|L]) :- paselt(T,L), tous_différents(L).
```

8. Résoudre le problème :

```
   S E N D
+  M O R E
-----------
 M O N E Y
```

où S, E, N, D, M, O, R et Y sont des chiffres deux à deux distincts.

```
somme([S,E,N,D,M,O,R,Y]) :-
   tous_chiffres([S,E,N,D,M,O,R,Y]),
   tous_différents([S,E,N,D,M,O,R,Y]),
   ((D + 10*N + 100*E + 1000*S) + (E + 10*R + 100*O + 1000*M)) =:=
     (Y + 10*E + 100*N + 1000*O + 10000*M).
```

Solution :

```
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2
```

9. Écrire un prédicat qui prend en entrée un élément et deux listes et qui est vrai si la deuxième liste peut être obtenue par une quelconque insertion de l'élément dans la première liste.

```
insère(T,M,[T|M]).
insère(T,[M|N],[M|Q]):- insère(T,N,Q).
```

10. Écrire un prédicat qui prend en entrée deux listes et qui est vrai si une liste est permutation des éléments de l'autre liste.

```
permutation([],[]).
permutation([T|R],L) :- permutation(R,M), insère(T,M,L).
```

11. Écrire une nouvelle solution du problème

```
   S E N D
+  M O R E
-----------
 M O N E Y
```

avec le prédicat sur les permutations.

```
tous_chiffres_différents(L) :- permutation([0,2,3,4,5,6,7,8,9],[X|[Y|L]]).
somme([S,E,N,D,M,O,R,Y]) :- tous_chiffres_différents([S,E,N,D,O,R,Y]), M = 1,
                            ((D + 10*N + 100*E + 1000*S) + (E + 10*R + 100*O + 1000*M)) =:=
                            (Y + 10*E + 100*N + 1000*O + 10000*M).
```

9 TD 9 : programmes Prolog et coupure

1. Écrivez, sans utiliser la primitive de division, un prédicat `division(A,B,Q,R)` où `Q` et `R` sont respectivement le quotient et le reste de la division de `A` par `B` (pour l'utilisation du prédicat, on suppose que `A` et `B` sont donnés, et que l'on recherche `Q` et `R`).

```
division(A,B,Q,R) :- Bmoins1 is B-1, between(0,Bmoins1,R), between(0,A,Q), A =:= B*Q+R.
```

Autre solution :

```
division(A,B,0,A):- A<B.
division(A,B,Q,R):- A >= B, BmoinsA is A-B, division(BmoinsA,B,Qmoins1,R), Q is Qmoins1 + 1.
```

2. Écrivez un prédicat qui prend en entrée un élément et une liste et qui est vrai si l'élément appartient à la liste. Optimisez l'exécution de ce prédicat au moyen d'une coupure.

```
appartient(X,[X|_]) :- !.
appartient(X,[_|R]) :- appartient(X,R).
```

3. Écrivez un prédicat qui prend en entrée un élément et une liste et qui est vrai si l'élément appartient au moins deux fois à la liste.

```
appartientbis(X,[X|R]) :- !, appartient(X,R).
appartientbis(X,[_|R]) :- appartientbis(X,R).
```

4. Écrivez un prédicat `somme_cube(A,B,C)` qui est vrai si l'entier `C` est la somme des cubes des entiers strictement positifs `A` et `B`. Écrivez ce prédicat de telle sorte qu'il puisse, étant donné `C`, énumérer toutes les valeurs possibles de `A` et de `B`.

```
somme_cube(A,B,C) :- between(1,C,A), between(1,C,B), C is A*A*A+B*B*B.
```

5. Optimisez le prédicat de la question 4, sachant que l'on n'a plus besoin que d'une seule solution.

```
somme_cube(A,B,C) :- between(1,C,A), between(1,C,B), C is A*A*A+B*B*B, !.
```

6. Optimisez le prédicat de la question 4, sachant que l'on ne veut pas perdre de solutions.
Note : cet exercice n'est pas très pédagogique...

```
somme_cube(A,B,C) :- s_cube(1,1,A,B,C).
s_cube(CA,_,_,_,C) :- 2*CA*CA*CA  > C, !, fail.
s_cube(CA,CB,A,B,C) :- CA*CA*CA+CB*CB*CB > C, !, NCA is CA+1, s_cube(NCA,NCA,A,B,C).
s_cube(CA,CB,A,B,C) :- C =\= CA*CA*CA+CB*CB*CB, !, NCB is CB+1, s_cube(CA,NCB,A,B,C).
s_cube(A,B,A,B,C) :-  C is A*A*A+B*B*B.
s_cube(CA,_,A,B,C) :- NCA is CA+1, s_cube(NCA,NCA,A,B,C).
```

Autre solution :

```
/* partout B>=A*/
somme_cube_opti(A,B,C):- scan(1,1,A,B,C).
scan(CA,CB,A,B,C):- CA*CA*CA+CB*CB*CB < C, ! ,
                    NCB is CB+1,
                    scan(CA,NCB,A,B,C). /* on parcourt avec le CB suivant*/
scan(CA,CB,A,B,C):- C is CA*CA*CA+CB*CB*CB, A is CA, B is CB.
        /* pas de coupure : on accepte les autres solutions */
scan(CA,CB,A,B,C):- CA*CA*CA+CB*CB*CB >= C, 2*CA*CA*CA<C, !,
        /* ">=" si égalité, on a une solution mais on continue */
                     NCA is CA+1,
                     NCB is NCA,
                     scan(NCA,NCB,A,B,C).
```

7. Écrivez un prédicat qui calcule le plus petit entier qui peut s'écrire, de deux manières différentes, comme la somme des cubes de deux entiers (ce nombre est inférieur à 10000).

```
plus_petit_double_cube(C) :-
   between(1,10000,C),somme_cube(A,_,C), somme_cube(D,_,C), A<D, !.
```

8. **Le motard généreux.** Un motard veut offrir une moto identique à chacune de ses petites amies. Le nombre représentant le coût de l'opération est, si l'on inverse tous les chiffres, celui qui représente le prix unitaire d'une moto. De mauvaises langues ont prétendu que le motard en question avait au moins deux petites amies, mais pas plus de 8. Le prix de la moto offerte se situant entre 10 000 et 99 999 francs, combien le motard a-t-il de petites amies et quel est le prix d'une moto ?

```
motard(F,M) :-
   between(2,8,F), between(1,9,A),between(0,9,B),between(0,9,C),
   between(0,9,D),between(0,9,E),
   M is A * 10000 + B * 1000 + C * 100 + D * 10 + E,
   F * M  =:= E * 10000 + D * 1000 + C * 100 + B * 10 + A.

F = 4
M = 21978 ;
```

10 TD 10 : révisions

Caml

1. Si l'expression suivante est bien formée, donnez son type :

```
#exception Erreur of int;;

#let f a b c =
   let rec g aa bb cc =
      let d = if aa > 0 then cc
         in if aa > bb then (raise (Erreur bb))
                       else g (aa+1) bb cc
            in try g a b c with Erreur t -> t;;
#f;;
- : int -> int -> unit -> int = <fun>
```

   Explications :
   - *aa > bb donc aa et bb sont de même type.*
   - *aa > 0 donc aa est de type entier, tout comme bb, a et b.*
   - *if aa > 0 then cc : pas de else, donc cc est de type unit, et donc c est de type unit.*
   - *Erreur t : t est de type entier, donc try g a b c with Erreur t -> t renvoie un entier.*
   - *On vérifie ensuite qu'il n'y a pas d'incohérences.*

2. Écrivez une fonctionnelle qui réalise la composition de deux fonctions (à un seul argument).

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

3. Écrivez une fonction polymorphe qui calcule la somme des valeurs d'une fonction `f` sur les N premiers entiers ($\sum_{i=1}^{N} \mathtt{f}(i)$). (Ne pas utiliser de boucle `for`.) Donnez le type de votre fonction.

```
#let rec somme f add n =if (n<=0)
                        then failwith "le troisième argument doit être supérieur ou égal à 1"
                        else match n with
                               1 -> f 1
                             | n -> add (somme f add (n-1)) (f n);;
val somme : (int -> 'a) -> ('a -> 'a -> 'a) -> int -> 'a = <fun>
```

4. Écrivez, au moyen d'une fonctionnelle sur listes, une fonction qui calcule le produit des éléments d'une liste d'entiers et qui renvoie 0 **dès** qu'un « 0 » est trouvé dans la liste.

```
#exception Zéro;;
exception Zéro

#let produit l = let mul x y = match (x,y) with   (0,_) -> raise Zéro
                                                 | (_,0) -> raise Zéro
                                                 | (x,y) -> x * y
 in try  (List.fold_left mul 1 l) with Zéro -> 0;;
val produit : int list -> int = <fun>
```

5. Écrivez, au moyen d'une fonctionnelle sur listes, une fonction qui prend en entrée une liste de caractères et qui construit la chaîne de caractères correspondante. On suppose que l'on a à notre disposition une fonction `cons` qui ajoute un caractère en début d'une chaîne.

```
#let cons c s = (String.make 1 c)^s;;
val cons : char -> string -> string = <fun>

#let concat l = List.fold_right cons l "";;
val concat : char list -> string = <fun>

#concat ['a';'b';'c'];;
- : string = "abc"
```

Prolog

1. **N-reines.** Le problème est de placer N reines sur un échiquier de telle sorte qu'il n'y ait pas deux reines sur une même ligne, colonne ou diagonale. Écrire un prédicat qui permette de construire une liste de N places sur un échiquier carré de taille N.

```
deuxindep(X,Y,U,V) :- X =\= U, Y =\= V, X+Y =\= U+V, X-Y =\= U-V.

tousindep(_,_,[]).
tousindep(X,Y,[[U,V]|L]) :- deuxindep(X,Y,U,V), tousindep(X,Y,L).

indep(_,0,S,S).
indep(N,M,L,S) :- between(1,N,X), between(1,N,Y), tousindep(X,Y,L),
                  P is M-1, indep(N,P,[[X,Y]|L],S).

nreines(N,L) :- indep(N,N,[],L).
```

   Autre solution :

```
indep(_,0,S,S) :-!.
indep(N,M,L,S) :- between(1,N,Y), tousindep(M,Y,L),
                  P is M-1, indep(N,P,[[M,Y]|L],S).
```

2. Même question mais en utilisant des négations par l'échec pour proscrire les mauvaises configurations.

```
deuxindep(X,_,X,_) :- !, fail.
deuxindep(_,Y,_,Y) :- !, fail.
deuxindep(X,Y,U,V) :- X+Y =:= U+V, !, fail.
deuxindep(X,Y,U,V) :- X-Y =:= U-V, !, fail.
deuxindep(_,_,_,_).
```

11 TP 1 : listes et récursivité

Fonctions classiques sur les ensembles triés

On considère des ensembles triés représentés par des listes. Exemple

```
#let l1 = [1;3;5;7;9];;
val l1 : int list = [1; 3; 5; 7; 9]

#let l2 = [2;3;6;8;9];;
val l2 : int list = [2; 3; 6; 8; 9]
```

Écrire des fonctions réalisant :

1. L'intersection ;

```
#let rec intersection l1 l2 = match l1,l2 with t1::r1,t2::r2-> if t1=t2
                                                                then t1::(intersection r1 r2)
                                                                else if t1<t2
                                                                     then (intersection r1 l2)
                                                                     else (intersection l1 r2)
                                                  |      _      -> [];;
val intersection : 'a list -> 'a list -> 'a list = <fun>
```

   Exemple :

```
#intersection l1 l2;;
- : int list = [3; 9]
```

2. L'union ;

```
#let rec union l1 l2 = match (l1,l2) with t1::r1,t2::r2 -> if t1=t2
                                                          then t1::(union r1 r2)
                                                          else if t1<t2
                                                               then t1::(union r1 l2)
                                                               else t2::(union l1 r2)
```

```
                                        |    [],l2     -> l2
                                        |    l1,[]     -> l1;;
val union : 'a list -> 'a list -> 'a list = <fun>
```

Exemple :

```
#union l1 l2;;
- : int list = [1; 2; 3; 5; 6; 7; 8; 9]
```

3. La différence ;

```
#let rec différence l1 l2 = match l1,l2 with t1::r1,t2::r2 ->if t1=t2
                                                   then (différence r1 r2)
                                                   else if t1<t2
                                                        then t1::(différence r1 l2)
                                                        else (différence l1 r2)
                                          |    l1,[]     -> l1
                                          |      _       -> [];;
val différence : 'a list -> 'a list -> 'a list = <fun>
```

Exemple :

```
#différence l1 l2;;
- : int list = [1; 5; 7]
```

4. La différence symétrique ;

```
#let différence_symétrique l1 l2 = différence (union l1 l2) (intersection l1 l2);;
val différence_symétrique : 'a list -> 'a list -> 'a list = <fun>
```

Exemple :

```
#différence_symétrique l1 l2;;
- : int list = [1; 2; 5; 6; 7; 8]
```

5. Le produit cartésien (résultat donné dans l'ordre lexicographique).

```
#let rec concatene l1 l2 = match l1 with   []   -> l2
                                        | t::r -> t :: concatene r l2 ;;
val concatene : 'a list -> 'a list -> 'a list = <fun>

#let rec produit_cartésien l1 l2 = match (l1,l2) with
                                       [t1],t2::r2 -> (t1,t2) :: produit_cartésien [t1] r2
                                    | t1::r1,t2::r2 -> concatene (produit_cartésien [t1] l2)
                                                                  (produit_cartésien  r1  l2)
                                    |       _       -> [];;
val produit_cartésien : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

Exemple :

```
#produit_cartésien l1 l2;;
- : (int * int) list =
[(1, 2); (1, 3); (1, 6); (1, 8); (1, 9); (3, 2); (3, 3); (3, 6); (3, 8);
(3, 9); (5, 2); (5, 3); (5, 6); (5, 8); (5, 9); (7, 2); (7, 3); (7, 6);
(7, 8); (7, 9); (9, 2); (9, 3); (9, 6); (9, 8); (9, 9)]
```

12 TP 2 : arbres n-aires et fonctionnelles

Arbres n-aires

Un arbre n-aire est soit un nœud interne (composé d'un entier, la clef, et d'une liste de sous-arbres), soit une feuille, c'est-à-dire un nœud sans fils. Nous définissons un arbre n-aire par le type suivant :

```
#type arbre = Noeud of (int * arbre list);;
type arbre = Noeud of (int * arbre list)
```

Fonctions élémentaires sur les arbres

1. Écrire une fonction `make_arbre` qui construit un arbre à partir d'un entier et d'une liste de sous-arbres.

```
# let make_arbre clé fils = Noeud (clé, fils);;
val make_arbre : int -> arbre list -> arbre = <fun>
```

2. Écrire des fonctions `clé` et `fils` qui extraient respectivement d'un nœud interne sa clé et la liste de ses fils.

```
# let clé  = function Noeud (c, _) -> c;;
val clé : arbre -> int = <fun>

# let fils = function Noeud (_, f) -> f;;
val fils : arbre -> arbre list = <fun>
```

3. Définissez une fonction `feuille` qui renvoie le booléen `true` si et seulement si son argument est une feuille.

```
# let feuille = function    Noeud(_,[]) -> true
                          |      _      -> false;;
val feuille : arbre -> bool = <fun>
```

Calcul sur les arbres

1. Définissez une fonction qui calcule la hauteur d'un arbre.

```
# let rec hauteur = function Noeud (_,f) -> 1 + (List.fold_right
                                                  max
                                                  (List.map hauteur f)
                                                  0);;
val hauteur : arbre -> int = <fun>

# let rec hauteur = function Noeud (_,f) -> 1+(List.fold_right
                                               (function x-> function y-> max (hauteur x) y)
                                               f
                                               0);;
val hauteur : arbre -> int = <fun>
```

2. Construisez les fonctions `nb_feuilles` et `nb_noeuds` qui calculent respectivement le nombre de feuilles et de nœuds (nœuds internes et feuilles) d'un arbre.

```
# let rec nb_feuilles = function
      Noeud (_,[]) -> 1
    | Noeud (_, f) -> List.fold_right
                          (function x -> function y-> x + y)
                          (List.map nb_feuilles f)
                          0;;
val nb_feuilles : arbre -> int = <fun>

# let rec nb_feuilles = function
    Noeud (_,[]) -> 1
  | Noeud (_, f) -> List.fold_right
                        (function x -> function y-> (nb_feuilles x) + y)
                        f
                        0;;
val nb_feuilles : arbre -> int = <fun>

# let rec nb_noeuds = function
      Noeud (_,[]) -> 1
    | Noeud (_, f) -> 1 + (List.fold_right
                               (function x -> function y-> x + y)
                               (List.map nb_noeuds f)
                               0);;
val nb_noeuds : arbre -> int = <fun>

# let rec nb_noeuds = function
    Noeud (_,[]) -> 1
  | Noeud (_, f) -> 1 + (List.fold_right
                           (function x-> function y-> (nb_noeuds x) + y)
```

```
                              f
                              0);;
val nb_noeuds : arbre -> int = <fun>
```

Parcours dans les arbres

1. Écrivez une fonction `parcours_profondeur` qui énumère les clefs d'un arbre en profondeur d'abord.

```
# let rec parcours_profondeur = function
     Noeud (c, [])   -> print_int c
   | Noeud (c, fils) -> print_int c; List.iter parcours_profondeur fils;;
val parcours_profondeur : arbre -> unit = <fun>
```

```
# let rec parcours_profondeur = function Noeud(c, fils)-> print_int c;
                                                          List.iter parcours_profondeur fils;;
val parcours_profondeur : arbre -> unit = <fun>
```

2. Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié.

```
# let rec appartient n= function
     Noeud (c, [])   -> c = n
   | Noeud (c, fils) -> c = n or (List.fold_right
                                   (function x -> function y-> x or y)
                                   (List.map (appartient n) fils)
                                   false);;
val appartient : int -> arbre -> bool = <fun>
```

```
# let rec appartient n= function Noeud (c, fils) ->
    c = n or (List.fold_right
               (function x -> function y-> (appartient n x) or y)
               fils
               false);;
val appartient : int -> arbre -> bool = <fun>
```

3. Écrivez une fonction `arbres_égaux` qui teste l'égalité de deux arbres.

```
# let rec combine a b = match (a,b) with
      [] ,  []  -> []
   | t::r, u::s -> (t,u)::combine r s
   |   _ ,   _  -> failwith "Listes non combinables";;
val combine : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

```
# let rec arbres_égaux a b = match (a,b) with
     Noeud(c,[]),Noeud(d,[]) -> c = d
   | Noeud(c,[]),Noeud(d,_)  -> false
   | Noeud(c,_), Noeud(d,[]) -> false
   | Noeud(c,e), Noeud(d,f)  ->  List.fold_right
                                  (function (g,h) -> function y -> (arbres_égaux g h) & y)
                                  (combine e f)
                                  true;;
val arbres_égaux : arbre -> arbre -> bool = <fun>
```

4. Écrivez une fonction `parcours_largeur` qui énumère les clefs d'un arbre en largeur d'abord.

```
# let rec parcours_largeur a=
    let rec parcours_liste = function
       (Noeud (c, fils))::l   -> print_int c; parcours_liste (l@fils)
     |           _            -> ()
    in  parcours_liste [a];;
val parcours_largeur : arbre -> unit = <fun>
```

13 TP 3 : arbres n-aires et exceptions

Arbres n-aires

Un arbre n-aire est soit une feuille (un entier), soit un nœud interne (composé d'un entier, la clef, et d'une liste de sous-arbres). Nous définissons un arbre n-aire par le type suivant :

```
# type arbre = Noeud of (int * arbre list);;
type arbre = Noeud of (int * arbre list)
```

1. Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié.

```
# let rec appartient n= function
  | Noeud (c, [])   -> c = n
  | Noeud (c, fils) -> c = n or  (List.fold_right
                                   (function x -> function y-> (appartient n x) or y)
                                   fils
                                   false);;
val appartient : int -> arbre -> bool = <fun>
```

2. Écrivez une fonction `appartient_opt`, nouvelle version de `appartient` optimisée par l'usage d'exceptions.

```
# exception Trouvé;;
exception Trouvé

# let appartient_opt n tree =
    let rec appartient_aux = function
       Noeud (c, [])   -> if c = n then raise Trouvé else false
     | Noeud (c, fils) -> if c = n then raise Trouvé
                                  else let l = (List.map appartient_aux fils) in false
    in try appartient_aux tree with Trouvé -> true;;
val appartient_opt : int -> arbre -> bool = <fun>
```

Exceptions

On se place de nouveau dans le cadre des arbres binaires définis par le type suivant :

```
# type arbre = Feuille of int | Noeud of (int * arbre  * arbre);;
type arbre = Feuille of int | Noeud of (int * arbre * arbre)
```

– Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié ; et qui se termine dès que l'élément recherché a été trouvé.

```
# exception Trouvé;;
exception Trouvé

# let appartient n tree=
   let rec appartient_aux = function
   Feuille f
      -> if (f = n) then raise Trouvé else false
   | Noeud (c, fg, fd)
       -> if c = n
          then  raise Trouvé
          else (appartient_aux fg) or (appartient_aux fd)
   in  try appartient_aux tree with Trouvé -> true;;
val appartient : int -> arbre -> bool = <fun>
```

– Écrivez une fonction `appartient_prof` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus la profondeur du nœud contenant l'élément recherché (on renverra 0 si l'élément recherché est absent de l'arbre).

```
# exception Trouvé_prof of int;;
exception Trouvé_prof of int

# let appartient_prof n tree=
   let rec appartient_prof_aux prof = function
```

```
   Feuille f
      -> if (f = n) then raise (Trouvé_prof prof) else false
   | Noeud (c, fg, fd)
      -> if c = n
         then  raise (Trouvé_prof prof)
         else (appartient_prof_aux (prof+1) fg) or
              (appartient_prof_aux (prof+1) fd)
   in  try (appartient_prof_aux 1 tree,0) with Trouvé_prof n -> (true,n);;
val appartient_prof : int -> arbre -> bool * int = <fun>
```

– Écrivez une fonction `appartient_arbre` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus le sous-arbre dont la racine est l'élément recherché (on renverra l'arbre de départ si l'élément recherché en est absent).

```
# exception Trouvé_arbre of arbre;;
exception Trouvé_arbre of arbre

# let appartient_arbre n tree=
   let rec appartient_arbre_aux = function
   (Feuille f) as t
      -> if (f = n) then raise (Trouvé_arbre t) else false
   | (Noeud (c, fg, fd)) as t
      -> if c = n
         then  raise (Trouvé_arbre t)
         else (appartient_arbre_aux fg) or (appartient_arbre_aux fd)
   in  try (appartient_arbre_aux tree, tree)
       with Trouvé_arbre t -> (true, t);;
val appartient_arbre : int -> arbre -> bool * arbre = <fun>
```

– Écrivez une fonction `appartient_arbre_père` qui recherche un élément dans un arbre et qui renvoie le sous-arbre dont un des fils a l'élément recherché pour racine (si celui existe).

```
# exception Trouvé_arbre;;
exception Trouvé_arbre

# exception Trouvé_père of arbre;;
exception Trouvé_père of arbre

# let appartient_arbre_père n tree=
   let rec appartient_arbre_père_aux = function
   Feuille f
      -> if (f = n) then raise Trouvé_arbre else false
   | (Noeud (c, fg, fd)) as t
      -> if c = n
         then  raise Trouvé_arbre
         else   try
                  (appartient_arbre_père_aux fg) or
                  (appartient_arbre_père_aux fd)
                with Trouvé_arbre -> raise (Trouvé_père t)
   in  try
         if (appartient_arbre_père_aux tree)
         then failwith "Cas impossible"
         else failwith "Élément absent de l'arbre"
       with    Trouvé_père t -> t
             | Trouvé_arbre  -> failwith "Élément à la racine";;
val appartient_arbre_père : int -> arbre -> arb
re = <fun>
```

14 TP 4 : petits programmes Prolog

1. Calculer la longueur d'une liste.

```
size([],0).
```

```
size([T|R],S) :- size(R,U), S is U+1.
```

2. Calculer la somme des éléments d'une liste.

```
sum([],0).
sum([T|R],M) :- sum(R,S), M is T+S.
```

3. Calculer la moyenne des éléments d'une liste.

```
moyenne(L,M) :- sum(L,S), size(L,A), M is S/A.
```

4. On appelle carré parfait un carré à cinq chiffres, dont la somme des chiffres est égale à 27 et dont les chiffres des milliers, des dizaines, des unités et des centaines sont consécutifs (et croissants dans l'ordre donné).

```
chiffre(X) :- between(0,9,X).

carre_parfait(X) :-
   chiffre(A), chiffre(B),
   D is B+1, E is D+1, C is E+1,
   X is A*10000 + B*1000 + C*100 + D*10  + E,
   A + B + C + D + E =:= 27,
   between(1,1000,Y),
   X =:= Y*Y.
```

5. Un carré magique est un tableau d'entiers positifs de trois cases de côtés dont la somme des lignes et des colonnes est constante et dont tous les éléments sont deux à deux distincts. Écrire un programme qui vérifie si un carré est magique de somme une valeur donnée.
   Exemple :

```
?- carré_magique(A,B,C,D,E,F,G,H,I,12).

A = 1
B = 3
C = 8
D = 5
E = 7
F = 0
G = 6
H = 2
I = 4

Yes

tous_différents([]).
tous_différents([T|L]) :- paselt(T,L), tous_différents(L).
paselt(_,[]).
paselt(X,[Y|L]) :- X =\= Y, paselt(X,L).
estelt(X,[X|_]).
estelt(X,[Y|L]) :- X =\= Y, estelt(X,L).
tous_nombres([],_).
tous_nombres([T|L],S) :- between(1,S,T), tous_nombres(L,S).
carré_magique(A,B,C,D,E,F,G,H,I,S) :-
tous_nombres([A,B,D,E],S),
C is S - A - B, C >= 0,
F is S - D - E, F >= 0,
G is S - A - D, G >= 0,
H is S - B - E, H >= 0,
I is S - G - H, I >= 0,
I is S - F - C,
tous_différents([A,B,C,D,E,F,G,H,I]).
```

15 TP 5 : décomposition d'un entier en facteurs premiers

Le but de ce TP est de réaliser la décomposition en facteurs premiers d'un entier. Ainsi, au nombre « 12 », nous voulons que ProLog associe la décomposition : « 2 * 2 * 3 ».

Dans tout le sujet nous supposerons que nous ne manipulons que des **entiers strictement positifs**.

1. Un nombre est premier s'il n'est divisible que par 1 et par lui-même. Écrivez un prédicat `nonpremier` qui est vrai si et seulement si le nombre qui lui ait passé en argument n'est pas premier, c'est-à-dire si et seulement si ce nombre admet un diviseur non trivial, ou est égal à un.

   ```
   nonpremier(1).
   nonpremier(X) :- Y is X-1, between(2,Y,Z), between(Z,Y,T), X =:= T*Z.
   ```

   Exemples d'utilisation :

   ```
   ?- nonpremier(1).
   Yes
   ?- nonpremier(7).
   No
   ?- nonpremier(12).
   Yes
   ```

2. Écrivez, à partir du prédicat `nonpremier`, un prédicat `premier` qui est vrai si et seulement si le nombre qu'il prend en argument est premier.

   ```
   premier(X) :- nonpremier(X), !, fail.
   premier(_).
   ```

   Exemples d'utilisation :

   ```
   ?- premier(1).
   No
   ?- premier(7).
   Yes
   ?- premier(12).
   No
   ```

3. Écrivez un prédicat `touslespremiers`, prenant en argument deux nombres `X` et `Y`, et vrai si `X` est un nombre premier inférieur ou égal à `Y`. Ce prédicat doit pouvoir énumérer tous les nombres premiers `X` inférieurs ou égaux à une valeur `Y` donnée.

   ```
   touslespremiers(X,Y):- between(2,Y,X), premier(X).
   ```

   Exemple d'utilisation :

   ```
   ?- touslespremiers(X,9).
   X = 2 ;
   X = 3 ;
   X = 5 ;
   X = 7 ;
   No
   ```

4. Écrivez, sans utiliser la primitive de division, un prédicat `division(A,B,Q,R)` où `Q` et `R` sont respectivement le quotient et le reste de la division de `A` par `B` (pour l'utilisation du prédicat, on suppose que `A` et `B` sont donnés, et que l'on recherche `Q` et `R`).

   ```
   division(A,B,Q,R) :-
   Bmoins1 is B-1, between(0,Bmoins1,R), between(0,A,Q), A =:= B*Q+R.
   ```

   autre solution :

```
division(A,B,0,A):- A<B.
division(A,B,Q,R):- A >= B, BmoinsA is A - B,
                    division(BmoinsA, B, Qmoins1, R),
                    Q is Qmoins1 + 1.
```

5. Écrivez, à partir du prédicat `division`, un prédicat `divise` prenant en entrée trois entiers `A`, `B` et `C` et qui est vrai si `A` est divisible par `B` de quotient `C`. Ce prédicat devra être capable de calculer `C` connaissant `A` et `B`.

   ```
   divise(A,B,C):-division(A,B,C,0).
   ```

   Exemple d'utilisation :

   ```
   ?- divise(12,2,C).
   C = 6
   Yes
   ```

6. Écrivez une fonction `decomposition` prenant en entrée deux variables `X` et `D` et qui, à `X` entier de valeur donnée, associe `D`, une de ses décompositions en facteurs premiers.

   ```
   decomposition(X,X):-premier(X),!.
   decomposition(X,D):-touslespremiers(Y,X), divise(X,Y,Z),
                       decomposition(Z,E), D = Y * E.
   ```

   Exemples d'utilisation :

   ```
   ?- decomposition(7,X).
   X = 7 ;
   No
   ?- decomposition(12,X).
   X = 2* (2*3) ;
   X = 2* (3*2) ;
   X = 3* (2*2) ;
   No
   ```

   Note : les parenthèses n'ont ici aucune espèce d'importance...

7. Modifiez le prédicat précédent pour qu'il ne produise plus qu'une décomposition en facteurs premiers, celle dans laquelle les facteurs sont rangés dans l'ordre croissant.

   ```
   decompositionbis(X,X):-premier(X),!.
   decompositionbis(X,D):-touslespremiers(Y,X), divise(X,Y,Z), !,
                          decompositionbis(Z,E), D = Y * E.
   ```

   Exemples d'utilisation :

   ```
   ?- decompositionbis(12,X).
   X = 2* (2*3) ;
   No
   ?- decompositionbis(2001,X).
   X = 3* (23*29) ;
   No
   ```

8. Écrivez un prédicat `pluspetitpremier` prenant en entrée deux variables `X` et `Y` et qui est vrai si et seulement si `X` est le plus petit entier premier supérieur ou égal à l'entier `Y` de valeur donnée.

   ```
   pluspetitpremier(X,X):- premier(X), !.
   pluspetitpremier(X,Z):- Y is Z+1, pluspetitpremier(X,Y).
   ```

Exemples d'utilisation :

```
?- pluspetitpremier(X,97).
X = 97 ;
No
?- pluspetitpremier(X,98).
X = 101 ;
No
```

16 Mini projet

Vecteurs d'entiers

Dans tout ce devoir, nous allons représenter les vecteurs par des listes. Ainsi, le vecteur à trois dimensions de coordonnées 1, 3 et 5, sera représenté par la liste : `[1 ;3 ;5]`.

Pour l'instant nous ne manipulons que des vecteurs d'entiers.

1. Écrivez une fonction prenant en entrée deux vecteurs et retournant le produit scalaire de ces vecteurs.

```
#let rec produit_scalaire x y =
   match x,y with
     a::b,c::d -> a*c + (produit_scalaire b d)
   |  [] , []  -> 0
   |     _     -> failwith "Vecteurs de tailles différentes";;
val produit_scalaire : int list -> int list -> int = <fun>
```

Type et exemple d'utilisation :

```
#produit_scalaire;;
- : int list -> int list -> int = <fun>

#produit_scalaire [1;3;5] [2;-6;1];;
- : int = -11
```

2. Écrivez une fonction prenant en entrée un vecteur `v` et une liste `l` de vecteurs, et renvoyant la liste des produits scalaires de `v` avec les éléments de `l`.

   (a) Écrivez une version récursive de cette fonction.

```
#let rec produit_vec_liste v = function
      []  -> []
    | t::r -> (produit_scalaire t v)::(produit_vec_liste v r);;
val produit_vec_liste : int list -> int list list -> int list = <fun>
```

   (b) Écrivez une version de cette fonction utilisant une ou des fonctionnelles prédéfinies, mais pas la récursivité.

```
#let produit_vec_liste v l =
      List.map (function x -> produit_scalaire x v) l;;
val produit_vec_liste : int list -> int list list -> int list = <fun>

#let produit_vec_liste v l = List.map (produit_scalaire v) l;;
val produit_vec_liste : int list -> int list list -> int list = <fun>
```

Type et exemple d'utilisation :

```
#produit_vec_liste;;
- : int list -> int list list -> int list = <fun>

#produit_vec_liste [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 0; 22]
```

3. Écrivez une fonction prenant en entrée deux listes `l` et `ll` de vecteurs, et renvoyant la liste des produits scalaires des éléments de `l` et de `ll`.

   (a) Écrivez une version récursive de cette fonction.

```
#let rec produit_vec_vec l = function
      []  -> []
   | t::r -> (produit_vec_liste t l)@(produit_vec_vec l r);;
val produit_vec_vec : int list list -> int list list -> int list = <fun>
```

(b) Écrivez une version de cette fonction utilisant une ou des fonctionnelles prédéfinies, mais pas la récursivité.

```
#let produit_vec_vec l ll =
    List.fold_left
       (function x -> function y -> x @ (produit_vec_liste y l))
       []
       ll;;
val produit_vec_vec : int list list -> int list list -> int list = <fun
>

#let produit_vec_vec l ll =
    List.fold_right
       (function x -> function y -> (produit_vec_liste x l) @ y)
       ll
       [];;
val produit_vec_vec : int list list -> int list list -> int list = <fun
>

#let produit_vec_vec l ll =
    let liste = List.map (function x -> (produit_vec_liste x l)) ll
    in  List.fold_left
           (function x -> function y -> x@y)
           []
           liste;;
val produit_vec_vec : int list list -> int list list -> int list = <f
un>
```

Type et exemple d'utilisation :

```
#produit_vec_vec;;
- : int list list -> int list list -> int list = <fun>

#produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

Des vecteurs d'un type quelconque

Écrivez une fonction calculant le produit scalaire de deux vecteurs d'un même type quelconque.

```
#let rec produit_scalaire_polym x y add neutre_add mul =
   match x,y with
     a::b,c::d -> (add (mul a c) (produit_scalaire_polym b d add neutre_add mul))
   |  [] , []  -> neutre_add
   |     _     -> failwith "Vecteurs de tailles différentes";;
val produit_scalaire_polym :
'a list -> 'b list -> ('c -> 'd -> 'd) -> 'd -> ('a -> 'b -> 'c) -> 'd =
<fun>
```

Dans toute la suite de ce devoir, nous manipulerons des vecteurs d'entiers.

Recherche d'un produit scalaire nul

Nous nous focalisons ici sur les couples de vecteurs de produit scalaire nul.

1. Écrivez une fonction `produit_scalaire_exc` qui prend en entrée deux vecteurs et qui renvoie leur produit scalaire, sauf si celui-ci est nul, auquel cas la fonction devra déclencher une exception `Nul`.

```
#exception Nul;;
exception Nul
```

```
#let produit_scalaire_exc x y =
    let ps = produit_scalaire x y in
    if  ps = 0 then raise Nul else ps;;
val produit_scalaire_exc : int list -> int list -> int = <fun>
```

Type et exemple d'utilisation :

```
#produit_scalaire_exc;;
- : int list -> int list -> int = <fun>

#produit_scalaire_exc [1;3;5] [2;-6;1];;
- : int = -11

#produit_scalaire_exc [1;3;5] [7;1;-2];;
Exception: Nul.
```

2. Écrivez une fonction `produit_vec_liste_exc v l` prenant en entrée un vecteur `v` et une liste `l` et qui renvoie la liste des produits scalaires du vecteur `v` avec les éléments de `l` sauf s'il existe un élément `u` de `l` dont le produit scalaire avec `v` est nul, auquel cas la fonction devra déclencher une exception `Vecteur` transportant le vecteur `u`.

   La fonction `produit_vec_liste_exc` devra utiliser la fonction `produit_scalaire_exc` et non la fonction `produit_scalaire`.

```
#exception Vecteur of int list;;
exception Vecteur of int list

#let rec produit_vec_liste_exc v = function
     []  -> []
  | t::r -> (try produit_scalaire_exc t v with Nul -> raise (Vecteur t))
            ::(produit_vec_liste_exc v r);;
val produit_vec_liste_exc : int list -> int list list -> int list = <fun>


#let produit_vec_liste_exc v l =
   List.fold_left
     (function x -> function y ->
       x@[(try produit_scalaire_exc y v with Nul -> raise (Vecteur y))])
     []
     l;;
val produit_vec_liste_exc : int list -> int list list -> int list = <
fun>
```

Type et exemple d'utilisation :

```
#produit_vec_liste_exc;;
- : int list -> int list list -> int list = <fun>

#produit_vec_liste_exc [1;3;5] [[2;-6;1]; [-4;2;4]];;
- : int list = [-11; 22]

#produit_vec_liste_exc [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Vecteur [7; 1; -2].
```

3. Écrivez une fonction `produit_vec_vec_exc` prenant en entrée deux listes de vecteurs, `l` et `ll`, et renvoyant la liste des produits scalaires des éléments de `l` avec ceux de `ll` sauf s'il existe un élément `u` de `l` et un élément `v` de `ll` dont le produit scalaire est nul, auquel cas la fonction devra déclencher l'exception `Paire` transportant la paire `(u,v)`.

   La fonction `produit_vec_vec_exc` devra utiliser la fonction `produit_vec_liste_exc` et non la fonction `produit_vec_liste`.

```
#exception Paire of int list * int list;;
exception Paire of int list * int list

#let rec produit_vec_vec_exc l = function
     []  -> []
  | t::r -> try (produit_vec_liste_exc t l)@(produit_vec_vec_exc l r)
            with Vecteur u -> raise (Paire (u,t));;
```

```
val produit_vec_vec_exc : int list list -> int list list -> int list = <f
un>
```

```
# let produit_vec_vec_exc l ll =
    List.fold_left
       (function x -> function y ->
          x@(try (produit_vec_liste_exc y l)
             with Vecteur u -> raise (Paire (u,y))))
       []
       ll;;
val produit_vec_vec_exc : int list list -> int list list -> int lis
t = <fun>
```

Type et exemple d'utilisation :

```
# produit_vec_vec_exc;;
- : int list list -> int list list -> int list = <fun>

# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;2;-2]; [-4;2;5]];;
- : int list = [-11; 3; 3; 5; 27; 1]

# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Paire ([1; 3; 5], [7; 1; -2]).
```

4. Finalement, écrivez une fonction `vecteurs_orthogonaux` prenant en entrée deux listes `l` et `ll` et renvoyant `([],[])`, sauf s'il existe un élément `u` de `l` et un élément `v` de `ll` dont le produit scalaire est nul, auquel cas la fonction renverra la paire `(u,v)`.

```
# let vecteurs_orthogonaux l ll =
    try let res = produit_vec_vec_exc l ll in ([],[])
    with Paire (u,v) -> (u,v);;
val vecteurs_orthogonaux :
int list list -> int list list -> int list * int list = <fun>

# let vecteurs_orthogonaux l ll =
    try produit_vec_vec_exc l ll ; ([],[]) with Paire (u,v) -> (u,v);;
# let vecteurs_orthogonaux l ll =
try produit_vec_vec_exc l ll ; ([],[]) with Paire (u,v) -> (u,v);;

Warning: this expression should have type unit.
val vecteurs_orthogonaux :
int list list -> int list list -> int list * int list = <fun>
```

Type et exemple d'utilisation :

```
# vecteurs_orthogonaux;;
- : int list list -> int list list -> int list * int list = <fun>

# vecteurs_orthogonaux [[1;3;5];[1;0;1]] [[2;-6;1];[7;2;-2];[-4;2;5]];;
- : int list * int list = [], []

# vecteurs_orthogonaux [[1;3;5];[1;0;1]] [[2;-6;1];[7;1;-2];[-4;2;4]];;
- : int list * int list = [1; 3; 5], [7; 1; -2]
```

De nouvelles fonctionnelles prédéfinies

Nous aimerions pouvoir écrire la fonction calculant le produit scalaire de deux vecteurs au moyen d'une des fonctionnelles prédéfinies vues en cours. Ce n'est pas possible, puisque nous avons besoin de décomposer simultanément deux listes. Voici d'autres fonctionnelles prédéfinies en `ocaml` :

1.
```
# List.iter2;;
- : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit = <fun>
```
`List.iter2 f [a1 ; ... ; an] [b1 ; ... ; bn]` est équivalent à
`f a1 b1 ; ... ; f an bn`.

2. `#List.map2;;`
   `- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>`
   `List.map2 f [a1 ; ... ; an] [b1 ; ... ; bn]` est équivalent à
   `[f a1 b1 ; ... ; f an bn]`.
3. `#List.fold_left2;;`
   `- : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a = <fun>`
   `List.fold_left2 f a [b1 ; ... ; bn] [c1 ; ... ; cn]` est équivalent à
   `f (... (f (f a b1 c1) b2 c2) ...) bn cn`.
4. `#List.fold_right2;;`
   `- : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c = <fun>`
   `List.fold_right2 f [a1 ; ... ; an] [b1 ; ... ; bn]` c est équivalent à
   `f a1 b1 (f a2 b2 (... (f an bn c) ...))`.

Toutes ces fonctionnelles déclenchent l'exception `Invalid_argument` si leur deux listes arguments ne sont pas de même taille.

1. Écrivez une fonction qui calcule le produit scalaire de deux vecteurs en utilisant une ou plusieurs des nouvelles fonctionnelles prédéfinies, mais sans utiliser la récursion.

```
#let produit_scalairef u v =
    let liste = List.map2 (function x -> function y -> x*y) u v
    in List.fold_left (function a -> function b -> a + b) 0 liste;;
val produit_scalairef : int list -> int list -> int = <fun>

#let produit_scalairef u v =
    List.fold_left2
       (function x -> function y -> function z -> x + y*z)
       0
       u
       v;;
val produit_scalairef : int list -> int list -> int = <fun>

#let produit_scalairef u v =
    List.fold_right2
       (function x -> function y -> function z -> x*y  + z)
       u
       v
       0;;
val produit_scalairef : int list -> int list -> int = <fun>
```

2. *Question facultative :* Écrivez une fonctionnelle `fold_left2` qui réalise (fait la même chose que) la fonctionnelle prédéfinie `List.fold_left2`.

```
#let rec fold_left2 f a b c = match (b,c) with
     [] , []  -> a
  | t::r,u::s -> fold_left2 f (f a t u) r s
  |     _     -> raise (Invalid_argument "fold_left2");;
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
=
<fun>
```

17 Examen de l'année 1999-2000

Note : le sujet comprenait le rappel des types et significations des fonctionnelles classiques sur les listes, ce rappel n'est pas repris ici (cf. le cours).

Exercice 1 : typage d'expressions `Ocaml`

Donnez le type des expressions suivantes.
Important : la justification du résultat trouvé comptera autant que le résultat lui-même !

1.
```
#let rec f g h x = if x > 0 then h x else g (h x);;
#f;;
- : ('a -> 'a) -> (int -> 'a) -> int -> 'a = <fun>
```
2.
```
#let f x y z =
     List.fold_right
        (function u -> function v -> if u=v then 2*(x u) else 3*(x u))
        y
        z;;
#f;;
- : (int -> int) -> int list -> int -> int = <fun>
```
3.
```
#exception Truc of float;;

#let f x y =
     let rec f_aux z =
        match z with
           t::r -> if t < y then raise (Truc t) else (f_aux r)
         |  _   -> failwith "Erreur"
     in try (f_aux x) with Truc n -> n;;
#f;;
- : float list -> float -> float = <fun>
```

Exercice 2 : élaboration de menus (en Ocaml)

Le but de ce problème est de construire un menu pour tous les repas de midi d'une semaine. Nous supposons disposer au départ d'une liste d'entrées, d'une liste de plats de résistance et d'une liste de desserts. Bien sûr, nous ne voulons pas que les menus prévus pour deux jours différents aient la même entrée, ou le même plat de résistance ou le même dessert.

Les fonctions nécessaires à la résolution de cet exercice seront naturellement polymorphes. C'est pourquoi l'énoncé ne précise pas le type utilisé pour représenter les entrées, les plats de résistance et les desserts (les trois sortes d'objets sont supposés de même type). Pour les exemples figurant dans l'énoncé, des chaînes de caractères ont été utilisées pour faciliter la compréhension.

1. **Construction des *menus*.**

   (a) Nous avons tout d'abord besoin d'une fonction qui prend en entrée une liste `l` d'éléments et qui renvoie la liste de toutes les listes qui contiennent un seul élément, cet élément appartenant à `l` (pour plus de clarté, voir l'exemple d'utilisation ci-dessous !).

   i. Écrivez une version récursive de cette fonction.
```
#let rec explose = function
      [] -> []
    | t::r -> [t]@(explose r);;
val explose : 'a list -> 'a list = <fun>
```
   ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.
```
#let explose l =
     List.map (function x -> [x]) l;;
val explose : 'a list -> 'a list list = <fun>
```
   Exemple d'utilisation :
```
#explose [1;2;4;5;6];;
- : int list list = [[1]; [2]; [4]; [5]; [6]]
```
   (b) Nous avons besoin d'une fonction qui prend en entrée un élément, `x`, et une liste de listes, `ll`, et qui rajoute `x` en tête de chacune des listes de `ll`.
```
#let rec étend_listes x = function
      [] -> []
    | t::r -> (x::t)::(étend_listes x r);;
val étend_listes : 'a -> 'a list list -> 'a list list = <fun>
```

```
# let étend_listes x l =
    List.map (function y -> x::y) l;;
val étend_listes : 'a -> 'a list list -> 'a list list = <fun>
```

Exemple d'utilisation :

```
# étend_listes 3 [[1;2];[4];[5;6]];;
- : int list list = [[3; 1; 2]; [3; 4]; [3; 5; 6]]
```

Écrivez une telle fonction.

(c) Nous avons besoin d'une fonction qui prend en entrée une liste, a, et une liste de listes, b, et qui construit la liste de toutes les listes obtenues en ajoutant un élément de a en tête d'une liste élément de b.

i. Écrivez une version récursive de cette fonction.

```
# let rec rallonge_listes a b = match a with
      [] -> []
    | t::r -> (étend_listes t b)@(rallonge_listes r b);;
val rallonge_listes : 'a list -> 'a list list -> 'a list list = <fun>
```

ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let rec rallonge_listes a b =
     List.fold_right (function x -> function y -> (étend_listes x b)@y)
                     a
                     [];;
val rallonge_listes : 'a list -> 'a list list -> 'a list list = <fun>
```

Exemple d'utilisation :

```
# rallonge_listes [1;2] [[3];[4;5]];;
- : int list list = [[1; 3]; [1; 4; 5]; [2; 3]; [2; 4; 5]]
```

(d) Écrivez une fonction qui prend en entrée trois listes, a, b et c, et qui construit la liste de toutes les listes [x ;y ;z] possibles à trois éléments telles que x est élément de a, y élément de b et z élément de c.

Par la suite nous appellerons *menu* une liste à trois éléments.

```
# let combine_trois_listes a b c =
    rallonge_listes a (rallonge_listes b (explose c));;
val combine_trois_listes : 'a list -> 'a list -> 'a list -> 'a list list =
<fun>
```

2. **Construction des *menus sur la semaine*.** Nous voulons considérer l'ensemble des menus de midi sur une semaine (cinq jours ouvrables). Par la suite nous appellerons *menu sur la semaine* une liste quelconque contenant exactement cinq *menus*. Écrivez donc une fonction qui prend en entrée la liste de tous les *menus* possibles et qui construit la liste de tous les *menus sur la semaine* possibles.

```
# let semaine l =
    rallonge_listes l (rallonge_listes l (rallonge_listes l (rallonge_listes
 l (explose l))));;
val semaine : 'a list -> 'a list list = <fun>
```

3. **Non redondance de deux *menus*.** Écrivez une fonction qui prend en entrée les *menus* de deux repas et qui indique si ces deux *menus* sont ou non redondants : deux *menus* sont dits redondants s'ils contiennent la même entrée, ou le même plat de résistance ou le même dessert. La fonction demandée doit renvoyer true si les deux *menus* arguments ne sont pas redondants, et elle doit renvoyer une exception Redondants dans le cas contraire.

```
# exception Redondants;;
exception Redondants

# let nonredondants m n =
    match (m,n) with
        ([a;b;c],[d;e;f]) -> if a=d or b=e or d=f
                             then raise Redondants
```

```
                              else true
      |        _          -> failwith "Mauvais arguments pour
 nonredondants";;
val nonredondants : 'a list -> 'a list -> bool = <fun>
```

Exemple d'utilisation :

```
# nonredondants
  ["Gougères aux deux céleris";"Filets de bondelle à la neuchâtelloise";
   "Rozenn à l'orange sauce mentholée"]
  ["Potage à la citrouille de Saint-Jacques-de-Montcalm";"Côte de boeuf rôtie à la
   bouquetière";"Demi-fraise à la maltaise"];;
- : bool = true

# nonredondants
  ["Soufflé de cervelle à la chanoinesse";"Aileron de dindonneau Sainte-Menehould";
   "Bordure de riz à la Montmorency"]
  ["Soufflé de cervelle à la chanoinesse";"Fricandeau de veau à l'oseille";
   "Croquets de Bar-sur-Aube"];;
Exception: Redondants.
```

4. **Non redondance d'un *menu* avec une liste de *menus*.** Nous avons besoin d'une fonction qui prend en entrée un *menu* `m` et une liste `l` de *menus*, et qui vérifie que le *menu* `m` n'est redondant avec aucun des *menus* de la liste `l`. Cette fonction doit impérativement utiliser la fonction `nonredondants` définie à la question précédente (question 3).

   (a) Nous voulons ici que notre fonction, appelée `sansredondances_exc`, déclenche une exception en cas de redondance, suivant le comportement défini par l'exemple d'utilisation.

      i. Écrivez une version récursive de cette fonction.

```
# let rec sansredondances_exc m = function
       []  -> true
    | t::r -> (nonredondants m t) & (sansredondances_exc m r);;
val sansredondances_exc : 'a list -> 'a list list -> bool = <fun>
```

      ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let sansredondances_exc m l =
      List.fold_right
         (function x -> function y -> y & (nonredondants m x))
         l
         true;;
val sansredondances_exc : 'a list -> 'a list list -> bool = <fun>
```

   Exemple d'utilisation :

```
# sansredondances_exc
  ["Gougères";"Filets de bondelle";"Rozenn"]
  [["Potage";"Côte de boeuf";"Demi-fraise"];
   ["Soufflé";"Dindonneau";"Bordure de riz"]];;
- : bool = true

# sansredondances_exc
  ["Gougères";"Filets de bondelle";"Rozenn"]
  [["Gougères";"Côte de boeuf";"Demi-fraise"];
   ["Soufflé";"Dindonneau";"Bordure de riz"]];;
Exception: Redondants.
```

   (b) Nous voulons ici que notre fonction, appelée `sansredondances_bool`, renvoie une valeur booléenne en cas de redondance, suivant le comportement défini par l'exemple d'utilisation. Comme précédemment, cette fonction doit impérativement utiliser la fonction `nonredondants` définie à la question 3.

      i. Écrivez une version récursive de cette fonction.

```
# let rec sansredondances_bool m = function
      [] -> true
    | t::r ->    (try (nonredondants m t) with Redondants -> false)
              & (sansredondances_bool m r);;
val sansredondances_bool : 'a list -> 'a list list -> bool = <fun>
```

ii. Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let sansredondances_bool m l =
    List.fold_right
       (function x -> function y ->
          (try (nonredondants m x) with Redondants -> false) & y)
       l
       true;;
val sansredondances_bool : 'a list -> 'a list list -> bool = <fun>
```

Exemple d'utilisation :

```
# sansredondances_bool
  ["Gougères";"Filets de bondelle";"Rozenn"]
  [["Potage";"Côte de boeuf";"Demi-fraise"];
   ["Soufflé";"Dindonneau";"Bordure de riz"]];;
- : bool = true
```

```
# sansredondances_bool
  ["Gougères";"Filets de bondelle";"Rozenn"]
  [["Gougères";"Côte de boeuf";"Demi-fraise"];
   ["Soufflé";"Dindonneau";"Bordure de riz"]];;
- : bool = false
```

5. **Non redondance d'un *menu sur la semaine***. Écrivez, à partir de sansredondances_bool, une fonction à valeurs booléennes qui prend en entrée un *menu sur la semaine* et qui est vrai si et seulement si ce *menu sur la semaine* n'est pas redondant.

```
# let menusansredondances = function
      []  -> true
    | t::r -> sansredondances_bool t r;;
val menusansredondances : 'a list list -> bool = <fun>
```

6. **Construction des *menus sur la semaine* non redondants**. Nous voulons une fonction qui prend en entrée la liste de tous les *menus sur la semaine* possibles et qui renvoie la liste de tous les *menus sur la semaine* non redondants.

   (a) Écrivez une version récursive de cette fonction.

```
# let rec semainesnonredondantes = function
      []  -> []
    | t::r -> if (menusansredondances t)
              then t::(semainesnonredondantes r)
              else (semainesnonredondantes r);;
val semainesnonredondantes : 'a list list list -> 'a list list list = <
fun>
```

   (b) Écrivez une version de cette fonction utilisant une des fonctionnelles prédéfinies.

```
# let semainesnonredondantes l =
    List.fold_right (function x -> function y ->
                         if (menusansredondances x) then x::y else y)
                    l
                    [];;
val semainesnonredondantes : 'a list list list -> 'a list list list = <
fun>
```

7. **La solution**. Écrivez une fonction qui prend en entrée la liste de toutes les entrées, celle de tous les plats de résistances, et celle de tous les desserts, et qui renvoie la liste de tous les *menus sur la semaine* possibles et non redondants.

```
# let touslesmenusnonredondants a b c =
    semainesnonredondantes (semaine (combine_trois_listes a b c));;
val touslesmenusnonredondants :
'a list -> 'a list -> 'a list -> 'a list list list = <fun>
```

Exercice 3 : élaboration de menus (en SWI-Prolog)

1. Écrivez un prédicat qui vérifie si un élément appartient à une liste.

```
appartient(X,[X|_]).
appartient(X,[_|R]):-appartient(X,R).
```

2. Optimisez la fonction de la question 1 par l'utilisation de coupure.

```
appartient(X,[X|_]):-!.
appartient(X,[_|R]):-appartient(X,R).
```

3. **Construction des *menus*.** Nous voulons un prédicat qui vérifie si une liste représente un menu. Écrivez un prédicat qui prend en entrée quatre listes, notées ici `X`, `Y`, `Z` et `T`, et qui est vrai si et seulement si la liste `T` est une liste à trois éléments, le premier appartenant à la liste `X`, le deuxième à la liste `Y` et le troisième à la liste `Z`.

```
menu(X,Y,Z,[A,B,C]):-appartient(A,X), appartient(B,Y), appartient(C,Z).
```

   Par la suite on appellera *menu* une liste pour laquelle ce prédicat est vrai.

4. **Construction des *menus sur la semaine*.** Nous voulons un prédicat qui vérifie si une liste est un *menu sur la semaine*, c'est-à-dire si une liste est une liste d'exactement cinq *menus*.
   Écrivez un prédicat qui prend en entrée quatre listes, les trois premières correspondants, dans l'ordre, aux listes des entrées, plats de résistance et desserts. Ce prédicat doit être vrai si et seulement si la quatrième liste contient exactement cinq éléments et que chacun de ces éléments est un *menu*.

```
menussurlasemaine(X,Y,Z,[A,B,C,D,E]):-
   menu(X,Y,Z,A),menu(X,Y,Z,B),menu(X,Y,Z,C),menu(X,Y,Z,D),menu(X,Y,Z,E).
```

5. **Non redondance de deux *menus*.** Nous voulons un prédicat `nonredondants` prenant en entrée deux *menus* et qui est vrai si et seulement si ces deux *menus* ne sont pas redondants, c'est-à-dire si et seulement si ces deux *menus* ne contiennent pas la même entrée, ni le même plat de résistance, ni le même dessert.

   (a) Écrivez le prédicat `nonredondants`.

```
nonredondants([A,B,C],[D,E,F]) :- A\=D,B\=E,C\=F.
```

   (b) Écrivez un prédicat `redondants` qui est vrai si deux *menus* sont redondants.

```
redondants([A,_,_],[D,_,_]).
redondants([_,B,_],[_,E,_]).
redondants([_,_,C],[_,_,F]).
```

   (c) Optimisez le prédicat de la question 5b par l'utilisation de coupure.

```
redondants([A,_,_],[D,_,_]):- !.
redondants([_,B,_],[_,E,_]):- !.
redondants([_,_,C],[_,_,F]).
```

(d) Écrivez une nouvelle version du prédicat `nonredondants`, cette version devra calculer sa valeur de retour en fonction de la valeur du prédicat `redondants` (usage de la négation).

```
nonredondants(A,B) :- redondants(A,B), !, fail.
nonredondants(A,B).
```

6. **Non redondance d'un *menu sur la semaine*.**

(a) Écrivez un prédicat qui vérifie qu'un *menu* n'est pas redondant avec une liste de *menus* (ce prédicat vérifie que le *menu* en question n'est redondant avec aucun des *menus* de la liste).

```
menuetliste(A,[]).
menuetliste(A,[T|R]):- nonredondants(A,T),menuetliste(A,R).
```

(b) Écrivez un prédicat qui est vrai si et seulement si une liste de *menus* ne contient pas de redondances.

```
listenonredondantes([]).
listenonredondantes([T|R]):- menuetliste(T,R),listenonredondantes(R).
```

7. **Obtention des *menus sur la semaine* non redondants.** Finalement, écrivez un prédicat qui prend en entrée quatre arguments, les trois premiers étant dans l'ordre une liste d'entrées, une de plats de résistance et une de desserts. Ce prédicat doit être vrai si et seulement si son quatrième argument est un *menu sur la semaine* non redondant.

```
menusurlasemainenonredondant(A,B,C,M):-
   menussurlasemaine(A,B,C,M),listenonredondantes(M).
```

8. **Obtention du premier *menu sur la semaine* non redondant.** Modifiez le prédicat précédent pour qu'il ne retourne que le premier *menu sur la semaine* non redondant trouvé.

```
menusurlasemainenonredondant(A,B,C,M):-
   menussurlasemaine(A,B,C,M),listenonredondantes(M),!.
```

18 Examen de l'année 1998-1999

18.1 Typage d'expressions ocaml

Donnez le type des expressions suivantes.

Important : la justification du résultat trouvé comptera autant que le résultat lui-même !

1.
```
#let f g x = g(g x);;
val f : ('a -> 'a) -> 'a -> 'a = <fun>
```

2.
```
#let rec f a b c =
    match a with
        []   -> [b c]
      | t::r  -> (b t)::(f r b c);;
val f : 'a list -> ('a -> 'b) -> 'a -> 'b list = <fun>
```

3.
```
#exception Problème;;
exception Problème

#let rec f g x y =
   try
     match y with
           []   -> if (g x) > x  then (g x)
                                 else raise Problème
         | t::r  -> f g (g t) r
   with Problème -> f g (x-1) y;;
val f : (int -> int) -> int -> int list -> int = <fun>
```

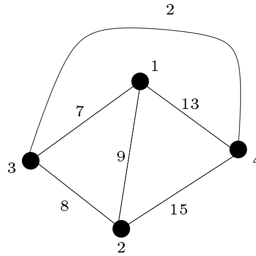


FIG. 1 – Exemple de problème avec quatre villes.

18.2 Les itérateurs sur liste en `ocaml`

Quatre itérateurs sur listes sont définis : `List.iter`, `List.map`, `List.fold_left` et `List.fold_right`. Les types de ces fonctionnelles et leurs comportements sont les suivants :
`List.iter : ('a -> 'b) -> 'a list -> unit`.
    `List.iter f [a1 ; ... ; an]` est équivalent à `begin f a1 ; ... ; f an ; () end`.
`List.map : ('a -> 'b) -> 'a list -> 'b list`.
    `List.map f [a1 ; ... ; an]` est équivalent à `[f a1 ; ... ; f an]`.
`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.
    `List.fold_left f a [b1 ; ... ; bn]` est équivalent à `f (...(f (f a b1) b2) ...) bn`.
`List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` .
    `List.fold_right f [a1 ; ... ; an] b` est équivalent à `f a1 (f a2 (... (f an b) ...))`.

Écrivez une fonction qui réalise `List.iter`, une qui réalise `List.map`, une qui réalise `List.fold_left` et une qui réalise `List.fold_right`. Les quatre fonctions demandées doivent bien évidemment être écrites en `ocaml`.

```
# let rec iter f = function
      []  -> ()
  | t::r -> begin f t; iter f r end;;
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
# let rec map f = function
      []  -> []
  | t::r -> (f t)::(map f r);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let rec fold_left f a = function
      []  -> a
  | t::r -> fold_left f (f a t) r;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# let rec fold_right f l b = match l with
      []  -> b
  | t::r -> f t (fold_right f r b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

18.3 Le voyageur de commerce (en `ocaml`)

Un voyageur de commerce doit visiter N villes. Il peut les parcourir dans l'ordre de son choix, du moment qu'il visite chaque ville exactement une fois et qu'il termine son parcours dans la ville de départ. Son problème est de minimiser la distance totale parcourue.

Les données du problème sont ici :

- Le nombre de villes, n. Les villes sont numérotées de 1 à n. ($n = 4$ dans l'exemple de la figure 1.)
- Une liste de distances entre les villes, notée ld. Chaque distance est décrite au moyen du type `chemin` :
  ```
  # type chemin = Chemin of (int * int list);;
  type chemin = Chemin of (int * int list)
  ```
  qui contient une paire constituée d'un entier (la longueur du chemin) et d'un liste d'entiers (la liste des villes visitées par le chemin, ordonnées dans l'ordre de visite). Exemple d'une valeur possible de ld dans

l'exemple de la figure 1 : $[Chemin(9,[1;2]); Chemin(8,[2;3]); Chemin(7,[1;3]); Chemin(13,[4;1]); Chemin(15,[4;2]); Chemin(2,[4;3])]$.

Conventions :

- Pour faciliter l'écriture des programmes, il est décidé que la liste d'entiers d'un objet de type `chemin` représente la liste des villes visitées, de la ville visitée en dernier (tête de liste) à la ville de départ (queue de la liste). Ainsi le chemin `Chemin(24,[1 ;4 ;2])` part de la ville 2, puis traverse la ville 4 avant d'aboutir à la ville 1.
- Pour alléger l'énoncé on appellera **chemin élémentaire** un objet de type `chemin` dont la liste d'entiers contient exactement deux entiers. Typiquement, ld est une liste de chemins élémentaires.

1. Écrivez une fonction `longueur` qui calcule la longueur d'une liste.

```
#let rec longueur = function
      []  -> 0
  | _::r -> 1 + (longueur r);;
val longueur : 'a list -> int = <fun>
```

2. Écrivez une fonction `nappartientpas` qui prend en entrée un objet et une liste et qui est vrai si et seulement si l'objet n'appartient pas à la liste.

```
#let rec nappartientpas x = function
      []  -> true
  | t::r -> (t <> x) & (nappartientpas x r);;
val nappartientpas : 'a -> 'a list -> bool = <fun>
```

3. Écrivez une fonction `debut` qui prend en entrée une liste de chemins élémentaires et qui retourne la liste de tous les chemins élémentaires possibles et commençant par la ville numéro 1.

Exemple : appelée sur la liste ld proposée ci-dessus, la fonction `debut` doit renvoyer la liste : `[Chemin(9,[2 ;1]) ; Chemin(7,[3 ;1]) ; Chemin(13,[4 ;1])]`.

```
#let rec debut = function
      []  -> []
  | (Chemin(long,[x;y]))::r ->
            let but = if x=1 then y else x in
                if ((x=1) or (y=1))
                   then (Chemin(long,[but;1]))::(debut r)
                   else (debut r)
  | _ -> failwith "debut";;
val debut : chemin list -> chemin list = <fun>
```

4. Écrivez une fonction `prolonge` qui prend en entrée un chemin (noté c) et un chemin élémentaire et qui prolonge c d'une ville, au moyen du chemin élémentaire, si c'est possible, c'est-à-dire si la tête de c est une des extrémités du chemin élémentaire et si les deux extrémités du chemin élémentaire ne sont pas déjà incluses dans c. La fonction `prolonge` réalise donc une sorte de concaténation de chemins.

```
#let prolonge c ce =
   match (c,ce) with
     (Chemin(l,t::r) , Chemin(le,[x;y])) ->
         let but = if x=t then y else x
            in if ((x=t) or (y=t)) & (nappartientpas but (t::r))
            then (Chemin(l+le,but::t::r))
            else failwith "pas prolongeable"
     | _ -> failwith "prolonge";;
val prolonge : chemin -> chemin -> chemin = <fun>
```

5. Écrivez une fonction `étend` qui prend en entrée un chemin (noté c) et une liste de chemins élémentaires (notée l) et qui construit la liste des chemins obtenus en prolongeant c d'une ville au moyen d'un chemin élémentaire de l. `étend` utilisera bien évidemment `prolonge`. Elle devra aussi mettre en œuvre un itérateur sur liste.

```
#let étend c l =
  let prolongateur chem liste =
        try (prolonge c chem)::liste
        with Failure "pas prolongeable" -> liste
```

```
  in List.fold_right prolongateur l [];;
val étend : chemin -> chemin list -> chemin list = <fun>
```

6. Écrivez une fonction `extension` qui prend en entrée une liste l de chemins et une liste le de chemins élémentaires et qui produit la liste de tous les chemins obtenus par prolongement d'un chemin de l au moyen d'un chemin de le.

```
#let extension l le =
  let extenseur chem liste = (étend chem le)@liste
   in List.fold_right extenseur l [];;
val extension : chemin list -> chemin list -> chemin list = <fun>
```

7. Écrivez une fonction `touslesschemins` qui prend en entrée n et ld et qui construit tous les chemins possibles contenant exactement n villes et commençant par la ville numéro 1.

```
#let tousleschemins n ld =
  let rec tousaux = function
        []  -> tousaux (debut ld)
     | ((Chemin(_,t))::r as l) -> if (longueur t) >= n
                                  then l
                                  else tousaux (extension l ld)
  in if n <= 1
     then failwith "tousleschemins"
     else tousaux [];;
val tousleschemins : int -> chemin list -> chemin list = <fun>
```

8. Maintenant que nous avons tous les chemins commençant par la ville numéro 1, nous voulons avoir tous les circuits possibles. Vous devez donc écrire une fonction `touslescircuits` construisant tous ces circuits.

   Attention : un détail interdit d'utiliser (directement ou indirectement) la fonction `prolonge` précédemment définie. Vous n'êtes pas obligés d'écrire des versions modifiées de fonctions existantes dont vous pourriez avoir besoin, vous pouvez vous contentez d'indiquer les différences entre les différentes versions.

```
#let prolongesoft c ce =
  match c,ce with
     Chemin(l,t::r), Chemin(le,[x;y]) ->
        let but = if x=t then y else x
           in if ((x=t) or (y=t)) & (but=1)
              then (Chemin(l+le,but::t::r))
              else failwith "pas prolongeable"
     | _ -> failwith "prolongesoft";;
val prolongesoft : chemin -> chemin -> chemin = <fun>

#let étendsoft c l =
  let prolongateur chem liste =
        try (prolongesoft c chem)::liste
        with Failure "pas prolongeable" -> liste
   in List.fold_right prolongateur l [];;
val étendsoft : chemin -> chemin list -> chemin list = <fun>

#let extensionsoft l le =
  let extenseur chem liste = (étendsoft chem le)@liste
   in List.fold_right extenseur l [];;
val extensionsoft : chemin list -> chemin list -> chemin list = <fun>

#let touslescircuits n ld =
     extensionsoft (tousleschemins n ld) ld;;
val touslescircuits : int -> chemin list -> chemin list = <fun>
```

9. Écrivez une fonction `plus_court` qui prend en entrée une liste de chemins et qui renvoie celui de plus courte distance.

```
#let plus_court l =
  let rec plus_court_aux pc = function
         []  -> pc
```

```
      | (Chemin(l,_) as chem)::r -> let (Chemin(lpc,_))=pc
                                    in if l < lpc
                                          then plus_court_aux chem r
                                          else plus_court_aux  pc  r
  in match l with
       []   -> failwith "plus_court"
     | tl::rl -> plus_court_aux tl rl;;
val plus_court : chemin list -> chemin = <fun>
```

10. Écrire une fonction qui résout le problème !

```
#let plus_court_circuit n ld =  plus_court(touslescircuits n ld);;
val plus_court_circuit : int -> chemin list -> chemin = <fun>
```

18.4 Le voyageur de commerce (en SWI-Prolog)

Un voyageur de commerce doit visiter N villes. Il peut les parcourir dans l'ordre de son choix, du moment qu'il visite chaque ville exactement une fois et qu'il termine son parcours dans la ville de départ. Son problème est de minimiser la distance totale parcourue.

Les données du problème sont ici :

- Le nombre de villes, N. ($N = 4$ dans l'exemple de la figure 1.)
- Une liste de distances entre les villes, notée LD. Chaque distance est décrite par une liste de trois entiers : le numéro de la ville de départ, celui de la ville d'arrivée et la distance séparant ces deux villes. Les villes sont numérotées de 1 à N. ($LD = [[1,2,9],[2,3,8],[1,3,7],[4,1,13],[4,2,15],[4,3,2]]$ dans l'exemple de la figure 1.)

1. Écrivez un prédicat `longueur_liste` qui calcule la longueur d'une liste.

```
longueur_liste([],0).
longueur_liste([_|R],L) :- longueur_liste(R,M), L is M+1.
```

2. Écrivez un prédicat `tous_elements` qui prend en entrée le nombre N de villes et qui vérifie que la liste ne contient que des entiers représentant des numéros de villes.

```
tous_elements([],_).
tous_elements([T|R],N) :- between(1,N,T), tous_elements(R,N).
```

3. Écrivez un prédicat `tous_differents` qui vérifie que les éléments d'une liste sont des nombres deux à deux distincts.

```
tous_differents([]).
tous_differents([T|R]):- pas_element(T,R), tous_differents(R).

pas_element(_,[]).
pas_element(X,[T|R]):- X =\= T, pas_element(X,R).
```

4. Écrivez un prédicat `dernier` qui permet d'extraire le dernier élément d'une liste.

```
dernier([X],X) :- !.
dernier([_|T],X) :- dernier(T,X).
```

5. Écrivez un prédicat `circuit` qui construit une liste de villes représentant le chemin que doit parcourir le voyageur de commerce (sans tenir compte pour l'instant de la minimisation).

```
circuit([T|R],N) :- longueur_liste(R,N), !, tous_elements(R,N),
                    tous_differents(R), dernier(R,T).
```

6. Écrivez un prédicat `distance` qui lit, dans la liste des distances LD, la distance entre deux villes données.

```
distance(X,Y,[[X,Y,D]|_],D) :- !.
distance(X,Y,[[Y,X,D]|_],D) :- !.
distance(X,Y,[[_,_,_]|R],L) :- distance(X,Y,R,L).
```

7. Écrivez un prédicat `longueur_circuit` qui calcule la longueur d'un circuit au moyen de la table des distances.

```
longueur_circuit([_],_,0).
longueur_circuit([X|[Y|R]],T,L) :-
   distance(X,Y,T,J), longueur_circuit([Y|R],T,K), L is J + K.
```

8. Écrivez un prédicat `existe_plus_petit` qui prend en entrée un circuit et qui est vrai si il existe un circuit plus court satisfaisant les hypothèses.

```
existe_plus_petit(C,T,N) :- longueur_circuit(C,T,L),
                            circuit(P,N), longueur_circuit(P,T,S),
                            S < L.
```

9. Écrivez un prédicat `plus_court_circuit` qui construit toutes les solutions au problème du voyageur de commerce.

```
plus_court_circuit(L,S,T,N) :- circuit(L,N), plus_court(L,T,N),
                              longueur_circuit(L,T,S).
plus_court(L,T,N) :- existe_plus_petit(L,T,N), !, fail.
plus_court(_,_,_).
```

10. Question subsidiaire : donnez au moins un exemple d'optimisation d'un des prédicats précédents au moyen de coupure.