Annales des TD/TP de programmation fonctionnelle et logique 1999

1 TD1 : récursivité et polymorphisme

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 $\mathrm{Fib}(n)$. Écrire deux versions de cette fonction : une qui utilise des tests (expressions booléennes) et l'autre le filtrage.

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

```
#let rec fib_filtre n =
 if n <= 1 then 1
           else (fib_filtre (n-1)) + (fib_filtre (n-2));;
val fib_filtre : int -> int = <fun>
```

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

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

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

3. Quelle est la complexité des différentes fonctions ?
Complexité de la première version :

$$C(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right].$$

Complexité de la deuxième version : $C(n) = n$.

Somme des éléments d'un tableau

1. Écrire une fonction qui prend en entrée un tableau et renvoie son élément maximal.

```
#let max t =
  let sup a b = if (a >= b) then a else b in
    let rec max_aux t i m =
      if i = -1
        then m
        else max_aux t (i-1) (sup t.(i) m) in
      max_aux t ((Array.length t)-1) t.((Array.length t)-1);;
val max : 'a array -> 'a = <fun>
```

Tri sur les entiers

1. Écrire une fonction qui prend en entrée un tableau t dont les $n-1$ premiers éléments sont triés et renvoie ce même tableau mais avec les n premiers éléments triés.

```
#let rec insère_aux t n =
  if n = 0
       then t
       else if t.(n) >= t.(n-1)
               then t
               else let temp = t.(n) in
                    t.(n)   <- t.(n-1);
                    t.(n-1) <- temp;
                    insère_aux t (n-1);;
val insère : 'a array -> int -> 'a array = <fun>
```

2. Au moyen de la fonction précédente, écrire une fonction de tri par insertion d'un tableau d'entiers.

```
#let tri t =
  let rec tri_aux t =
     function   0 -> t
              | n -> insère_aux (tri_aux t (n-1)) n in
     tri_aux t ((Array.length t)-1);;
val tri : 'a array -> 'a array = <fun>
```

Tri polymorphe

Écrire une fonction de tri polymorphe sur tableau.

```
#let tri ordre t =
  let rec insère_aux t n =
    if n = 0
         then t
         else if (ordre t.(n) t.(n-1))
                 then t
                 else let temp = t.(n) in
                      t.(n)   <- t.(n-1);
                      t.(n-1) <- temp;
                      insère_aux t (n-1); in
   let rec tri_aux t =
       function   0 -> t
```

```
              | n -> insère_aux (tri_aux t (n-1)) n in
     tri_aux t ((Array.length t)-1);;
val tri : ('a -> 'a -> bool) -> 'a array -> 'a array = <fun>
```

2 TD2 : typage et récursivité

Expressions à typer

1. `let f x y = x;;`
   `val f : 'a -> 'b -> 'a = <fun>`
2. `let f x y = y;;`
   `val f : 'a -> 'b -> 'b = <fun>`
3. `let f x y = if true then x else y;;`
   `val f : 'a -> 'a -> 'a = <fun>`
4. `let f x y = if false then x else y;;`
   `val f : 'a -> 'a -> 'a = <fun>`
5. `let f x y = if x=y then 0 else 1;;`
   `val f : 'a -> 'a -> int = <fun>`
6. `let f x y = if x=y then x else 1;;`
   `val f : int -> int -> int = <fun>`
7. `let f x y = if x=y then x+y else x-y;;`
   `val f : int -> int -> int = <fun>`
8. `let f x y = if x then x+y else x-y;;`
   `This expression has type bool but is here used with type int`

Petits programmes récursifs

1. Écrire une fonction récursive qui prend en entrée deux tableaux (supposés être de même taille) et qui vérifie si un tableau est le miroir de l'autre (ex. : `[|5;3;4;6|]` est le miroir de `[|6;4;3;5|]`).

```
# let miroir a b  =
   let rec miroir_aux rang =
      if rang = (Array.length a)
         then true
         else a.(rang)=b.((Array.length a)-1-rang) & miroir_aux (rang+1)
  in
   miroir_aux 0;;
val miroir : 'a array -> 'a array -> bool = <fun>
```

2. Écrire une fonction récursive qui prend en entrée deux tableaux (supposés être de même taille) et qui recopie le premier tableau dans le deuxième.

```
# let recopie origine destination =
   let rec recopie_aux rang =
      if rang = (Array.length destination)
         then destination
         else begin
                destination.(rang) <- origine.(rang);
                recopie_aux (rang+1)
              end
  in
   recopie_aux 0;;
val recopie : 'a array -> 'a array -> 'a array = <fun>
```

3 TD3 : récursivité et polymorphisme

Expressions à évaluer

Les expressions suivantes sont évaluées dans l'ordre. Quelles sont leurs valeurs respectives ?

```
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);;
- : 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
```

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>
```

Encore des fonctions opérant sur des tableaux

1. Écrire une fonction récursive qui calcul la somme des éléments d'un tableau entier de taille supérieure ou égale à un.

```
let somme tab =
  let taille = (Array.length tab)
  in let rec somme_aux rang =
    if (rang = taille -1)
      then tab.(rang)
      else tab.(rang) + (somme_aux (rang + 1))
    in if taille = 1
          then tab.(0)
          else tab.(0) + (somme_aux 1);;
val somme : int array -> int = <fun>
```

2. Écrire une fonction récursive qui calcul la somme des éléments d'un tableau de taille supérieure ou égale à un, quel que soit son type.

```
let somme add tab =
  let taille = (Array.length tab)
  in let rec somme_aux rang =
    if (rang = taille -1)
      then tab.(rang)
      else (add tab.(rang) (somme_aux (rang + 1)))
    in if taille = 1
          then tab.(0)
          else (add tab.(0) (somme_aux 1));;
val somme : ('a -> 'a -> 'a) -> 'a array -> 'a = <fun>
```

3. Typer la fonction précédente.
4. Écrire une fonction récursive qui applique une opération binaire (typiquement `+`,`*` ou `max`) aux éléments d'un tableau de taille supérieure ou égale à un, quel que soit son type.
   Réponse : la fonction écrite à la question 2 répond à la question.
5. Même question mais en ne supposant plus que le tableau est forcément de taille non nulle.

```
let application op neutre tab =
  let taille = (Array.length tab)
  in let rec application_aux rang =
    if (rang = taille)
      then neutre
      else (op tab.(rang) (application_aux (rang + 1)))
    in application_aux 0;;
val application : ('a -> 'b -> 'b) -> 'b -> 'a array -> 'b = <fun>
```

6. Typer la fonction précédente.

4 TD4 : listes et récursivité

Fonctions 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>
```

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>
```

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 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>
```

6. 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>
```

7. 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>
```

8. 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>
```

9. qui teste l'égalité de deux listes.

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

Tri par insertion générique

Le tri par insertion est un tri simple : à l'étape i il prend le i^e élément de la liste à trier et l'insère à la place idoine dans la liste des $i-1$ élément déjà triés.

1. Écrire une fonction réalisant le tri par insertion.

```
let rec tri_insertion =
let rec insère x = function
    []        -> [x]
```

```
  | t::r as l -> if x <= t
                  then x::l
                  else t::(insère x r)
in function
      []  -> []
   | t::r -> insère t (tri_insertion r);;
val tri_insertion : 'a list -> 'a list = <fun>
```

2. Réécrire cette fonction avec des **match** ... **with** et sans **function**.

```
let rec tri_insertion l =
let rec insère l x = match l with
    []        -> [x]
  | t::r as l -> if x <= t
                  then x::l
                  else t::(insère r x)
in match l with
      []  -> []
   | t::r -> insère (tri_insertion r) t;;
val tri_insertion : 'a list -> 'a list = <fun>
```

3. Passer le prédicat de test en paramètre.

```
let rec tri_insertion l comp =
let rec insère l x = match l with
    []        -> [x]
  | t::r as l -> if comp x t
                  then x::l
                  else t::(insère r x)
in match l with
      []  -> []
   | t::r -> insère (tri_insertion r comp) t;;
val tri_insertion : 'a list -> ('a -> 'a -> bool) -> 'a list = <fun>
```

4. Écrire le plus simplement possible une fonction de tri par ordre croissant et une fonction de tri dans ordre décroissant.

```
let tri_croissant l = tri_insertion l (function x -> function y -> x <= y);;
val tri_croissant : 'a list -> 'a list = <fun>

let tri_décroissant l = tri_insertion l (function x -> function y -> x >= y);;
val tri_décroissant : 'a list -> 'a list = <fun>
```

5 TP1 : la douloureuse découverte de ocaml

Fonctions classiques sur les ensembles triés

On considère des ensembles triés représentés par des listes. Écrire des procédures réalisant les fonctions suivantes :

1. 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>
```

2. 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>
```

3. Différence ;

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

4. Différence symétrique ;

```
let difference_symetrique l1 l2 =
  difference (union l1 l2) (intersection l1 l2);;
val difference_symetrique : 'a list -> 'a list -> 'a list = <fun>
```

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

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

Le tri fusion

Le tri fusion est un algorithme basé sur la paradigme « Diviser Pour Régner ». En effet, pour trier une liste d'éléments, on coupe la liste en deux listes de même longueur qu'on trie par tri fusion. On fusionne ensuite les deux listes ainsi triées en une seule liste elle aussi triée. Le tri d'une liste vide ou d'une liste à un seul élément étant évident.

Écrire le tri fusion au moyen d'une fonction **split** qui décompose une liste en deux sous-listes de longueur égales à une unité près en sans calculer de longueur de liste.

```
let rec split = function
   t1::t2::r -> let x,y = split r in t1::x,t2::y
 |    [t1]   -> [t1],[]
 |     []    -> [],[];;
val split : 'a list -> 'a list * 'a list = <fun>
```

```
let rec tri_fusion = function
    [] -> []
 | [x] -> [x]
 |  l  -> let x,y = split l in union (tri_fusion x) (tri_fusion y);;
val tri_fusion : 'a list -> 'a list = <fun>
```

6 TD5 et TP2 : Arbres

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 est défini par le type suivant :

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

Fonctions élémentaires sur les arbres

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

```
let make_arbre clé fils_gauche fils_droit =
   Noeud (clé, fils_gauche, fils_droit);;
val make_arbre : int -> arbre -> arbre -> arbre = <fun>
```

2. Écrire des fonctions `clé`, `fg` et `fd` qui extraient respectivement d'un nœud interne sa clé, son fils gauche et son fils droit.

```
let clé = function
         Feuille n     -> n
 | Noeud (c, f_g, f_d) -> c;;
val clé : arbre -> int = <fun>
let fg = function
    Noeud (c, f_g, f_d) -> f_g
  | _  -> failwith "fg";;
val fg : arbre -> arbre = <fun>
let fd = function
    Noeud (c, f_g, f_d) -> f_d
  | _  -> failwith "fd";;
val fg : arbre -> arbre = <fun>
```

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

```
let feuille = function
   Feuille _ -> 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
       Feuille _   -> 1
 | Noeud (_, fg, fd) -> 1 +  max (hauteur fg) (hauteur fd);;
 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
       Feuille _   -> 1
 | Noeud (_, fg, fd) -> (nb_feuilles fg) + (nb_feuilles fd);;
val nb_feuilles : arbre -> int = <fun>
let rec nb_noeuds = function
       Feuille _   -> 1
 | Noeud (_, fg, fd) -> 1 + (nb_noeuds fg) + (nb_noeuds fd);;
val nb_feuilles : 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
       Feuille f   -> print_int f
 | Noeud (c, fg, fd) -> print_int c ;
                       (parcours_profondeur fg);
                       (parcours_profondeur fd);;
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
       Feuille f   -> f = n
 | Noeud (c, fg, fd) -> c = n or
                            (appartient n fg) or
                            (appartient n fd);;
val appartient : int -> arbre -> bool = <fun>
```

3. 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
      (Feuille f)::l         -> print_int f; parcours_liste l
    |  (Noeud (c, fg, fd))::l -> print_int c; parcours_liste (l@[fg]@[fd])
    |           _              -> ()
  in
    parcours_liste [a];;
val parcours_largeur : arbre -> unit = <fun>
```

Arbres n-aires

Voir le TD suivant.

7 TD6 et TP3 : itérateurs et exceptions

Itérateurs sur arbres n-aires

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

Rappel sur les itérateurs

`List.iter : ('a -> unit) -> '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) ...)).

Fonctions élémentaires sur les arbres

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

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

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

```
let clé = function
         Feuille n     -> n
 | Noeud (c, f) -> c;;
val clé : arbre -> int = <fun>
let fils = function
    Noeud (c, f) -> f
  | _  -> failwith "fils";;
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
   Feuille _    -> true
 | 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
       Feuille _   -> 1
 | Noeud (_, f) -> 1 + (List.fold_right max
                                        (List.map hauteur f)
                                        (hauteur (List.hd f)));;
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
       Feuille _   -> 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_noeuds = function
       Feuille _   -> 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>
```

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
       Feuille f   -> print_int f
 | 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
       Feuille f   -> f = 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>
```

3. 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
      (Feuille f)::l         -> print_int f; parcours_liste l
   |  (Noeud (c, fils))::l   -> print_int c; parcours_liste (l@fils)
   |           _             -> ()
  in
    parcours_liste [a];;
val parcours_largeur : arbre -> unit = <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);;
```

– 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;;
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;;
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;;
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_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
        (appartient_arbre_père_aux tree); failwith "Élément absent de l'arbre"
   with    Trouvé_père t -> t
         | Trouvé_arbre  -> failwith "Élément à la racine";;
val appartient_arbre_père : int -> arbre -> arbre = <fun>
```

8 TD7 : Logique

Analyse de vérité

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

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

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

2. $a \to b = (\neg b) \to (\neg a)$ (raisonnement par contraposée).

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

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

<table>
<tr><td colspan="2">$a = 0$</td><td>$a = 1$</td></tr>
<tr><td colspan="2">$((0 \wedge b) \vee c) \leftrightarrow ((0 \vee c) \wedge (b \vee c))$
$(0 \vee c) \leftrightarrow (c \wedge (b \vee c))$
$c \leftrightarrow (c \wedge (b \vee c))$</td><td>$((1 \wedge b) \vee c) \leftrightarrow ((1 \vee c) \wedge (b \vee c))$
$(b \vee c) \leftrightarrow (1 \wedge (b \vee c))$
$(b \vee c) \leftrightarrow (b \vee c)$</td></tr>
<tr><td colspan="2">$c \leftrightarrow (c \wedge (b \vee c))$</td><td>1</td></tr>
<tr><td>$b = 0$</td><td>$b = 1$</td><td></td></tr>
<tr><td>$c \leftrightarrow (c \wedge (0 \vee c))$
$c \leftrightarrow (c \wedge c)$
$c \leftrightarrow c$
1</td><td>$c \leftrightarrow (c \wedge (1 \vee c))$
$c \leftrightarrow (c \wedge 1)$
$c \leftrightarrow c$
1</td><td></td></tr>
</table>

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

<table>
<tr><td>$a = 0$</td><td colspan="2">$a = 1$</td></tr>
<tr><td>$((0 \vee b) \wedge c) \leftrightarrow ((0 \wedge c) \vee (b \wedge c))$
$(b \wedge c) \leftrightarrow (0 \vee (b \wedge c))$
$(b \wedge c) \leftrightarrow (b \wedge c)$
1</td><td colspan="2">$((1 \vee b) \wedge c) \leftrightarrow ((1 \wedge c) \vee (b \wedge c))$
$(1 \wedge c) \leftrightarrow (c \vee (b \wedge c))$
$c \leftrightarrow (c \vee (b \wedge c))$
$c \leftrightarrow (c \vee (b \wedge c))$</td></tr>
<tr><td></td><td>$b = 0$</td><td>$b = 1$</td></tr>
<tr><td></td><td>$c \leftrightarrow (c \vee (1 \wedge c))$
$c \leftrightarrow (c \vee c)$
$c \leftrightarrow c$
1</td><td>$c \leftrightarrow (c \vee (1 \wedge c))$
$c \leftrightarrow (c \vee c)$
$c \leftrightarrow c$
1</td></tr>
</table>

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 suivants :

1.
$$((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t$$

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

$p \to (q \vee r) = (q \vee r) \vee \neg p = q \vee r \vee \neg p$
$(q \vee s) \to (t \wedge u) = (t \wedge u) \vee \neg(q \vee s) = (t \wedge u) \vee (\neg q \wedge \neg s) = (t \vee (\neg q \wedge \neg s)) \wedge (u \vee (\neg q \wedge \neg s)) =$
$(t \vee \neg q) \wedge (t \vee \neg s) \wedge (u \vee \neg q) \wedge (u \vee \neg s)$
D'où :

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

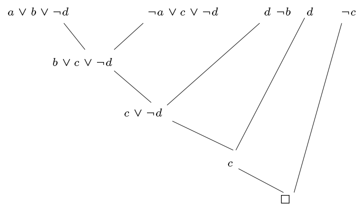
2.
$$(p \wedge q) \vee (\neg p \wedge \neg r) \to (q \leftrightarrow r)$$

$(p \wedge q) \vee (\neg p \wedge \neg r) \to (q \leftrightarrow r) =$
$(q \leftrightarrow r) \vee \neg((p \wedge q) \vee (\neg p \wedge \neg r)) =$
$((q \to r) \wedge (r \to q)) \vee (\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg r)) =$
$((r \vee \neg q) \wedge (q \vee \neg r)) \vee (\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg r)) =$
$((r \vee \neg q) \vee (\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg r))) \wedge ((q \vee \neg r) \vee (\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg r))) =$
$((r \vee \neg q) \vee ((\neg p \vee \neg q) \wedge (p \vee r))) \wedge ((q \vee \neg r) \vee ((\neg p \vee \neg q) \wedge (p \vee r))) =$
$(((r \vee \neg q) \vee (\neg p \vee \neg q)) \wedge ((r \vee \neg q) \vee (p \vee r))) \wedge (((q \vee \neg r) \vee (\neg p \vee \neg q)) \wedge ((q \vee \neg r) \vee (p \vee r))) =$
$(r \vee \neg q \vee \neg p \vee \neg q) \wedge (r \vee \neg q \vee p \vee r) \wedge (q \vee \neg r \vee \neg p \vee \neg q) \wedge (q \vee \neg r \vee p \vee r) =$
$(r \vee \neg q \vee \neg p) \wedge (r \vee \neg q \vee p) =$
$(r \vee \neg q)$

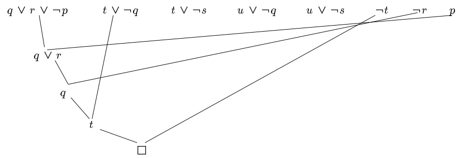
Méthode de résolution

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.
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.
3. Aucun membre de la direction de la bibiothèque ne peut être membre de la direction financière.

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.

1. $f \to g$
2. $(g \wedge b) \to f$
3. $b \to \neg f$

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 : $((f \to g) \wedge ((g \wedge b) \to f) \wedge (b \to \neg f)) \leftrightarrow ((f \to g) \wedge (g \to \neg b))$. La démonstration est établie par la table 1.
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 bibiothèque.

9 TD8 : premiers programmes ProLog

1. Écrire un prédicat « hypothenuse » qui prend trois arguments X, Y et Z et qui est vrai si $Z^2 = X^2 + Y^2$.
```
hypothenuse(X,Y,Z) :- X*X + Y*Y =:= Z*Z.
```
2. Écrire un prédicat qui est vrai si son argument est un chiffre.
```
estelt(X,[X|_]).
estelt(X,[_|L]) :- estelt(X,L).
chiffre(T) :- estelt(T,[0,1,2,3,4,5,6,7,8,9]).
```
3. Réécrire le prédicat « hypothenuse » pour qu'il puisse énumérer tous les triplets (X, Y, Z) satisfaisants la relation $X^2 + Y^2 = Z^2$.
```
hypothenuse(X,Y,Z) :- chiffre(X), chiffre(Y), chiffre(Z),
 X*X + Y*Y =:= Z*Z.
```
4. 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.
```
paselt(_,[]).
paselt(X,[Y|L]) :-  X =\= Y, paselt(X,L).
```
5. Écrire un prédicat qui est vrai si tous les éléments d'une liste sont différents.

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

TAB. 1: Analyse de vérité de la formule de trasncription.

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

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

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

Autre solution :

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

7. 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.

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

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

somme([S,E,N,D,M,O,R,Y]) :-
   tous_chiffres([S,E,N,D,M,O,R,Y]),
   tous_differents([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).
```

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

```
permutation([],[]).
permutation([T|R],L) :- permutation(R,M), insere(T,M,L).
insere(T,M,[T|M]):-
insere(T,[M|N],[M|Q]):- insere(T,N,Q).
```

9. 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_differents(L) :- permutation([0,2,3,4,5,6,7,8,9],[_|[_|L]]).
somme([S,E,N,D,M,O,R,Y]) :-
   tous_chiffres_differents([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).
```

10 TP4 : la douloureuse découverte de SWI-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(1,100000,Y),
   X =:= Y*Y.
```

5. Un carré magique est un tableau de trois cases de côtés dont la somme des lignes et des colonnes est constante. Écrire un programme qui vérifie si un carré est magique de somme une valeur donnée.

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

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

nombre(T) :- chiffre(X), chiffre(Y), chiffre(Z), T is X*100 + 10*Y + Z.

tous_nombres([],S).
tous_nombres([T|L],S) :- between(1,S,T), tous_nombres(L,S).

carre_magique(A,B,C,D,E,F,G,H,I,S) :-
tous_nombres([A,B,C,D,E,F,G,H,I],S),
S =:= A + B + C,
S =:= D + E + F,
S =:= G + H + I,
S =:= A + D + G,
S =:= B + E + H,
S =:= C + F + I,
tous_differents([A,B,C,D,E,F,G,H,I]).
```

6. Un carré magique est parfait si la somme de ses 6 diagonales est constante et égale à la somme de ses lignes.

```
carre_magique_parfait(A,B,C,D,E,F,G,H,I,S) :-
tous_nombres([A,B,C,D,E,F,G,H,I],S),
S =:= A + B + C,
S =:= D + E + F,
S =:= G + H + I,
S =:= A + D + G,
S =:= B + E + H,
S =:= C + F + I,
S =:= A + E + I,
S =:= B + F + G,
S =:= C + D + H,
S =:= G + E + C,
S =:= H + F + A,
S =:= D + B + I,
tous_differents([A,B,C,D,E,F,G,H,I]).
```

11 TD9 : encore des petits programmes Prolog

1. **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
```

2. Un carré magique est un tableau de trois cases de côtés dont chaque case contient un entier naturel, et dont la somme des lignes et des colonnes est constante. Écrire un programme qui vérifie si un carré est magique de somme une valeur donnée (inférieure à 100).

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

carre_magique(A,B,C,D,E,F,G,H,I,S) :-
between(0,100,S),
between(1,S,A),
between(1,S,B),
between(1,S,C),
between(1,S,D),
between(1,S,E),
between(1,S,F),
between(1,S,G),
between(1,S,H),
between(1,S,I),
S =:= A + B + C,
S =:= D + E + F,
```

```
S =:= G + H + I,
S =:= A + D + G,
S =:= B + E + H,
S =:= C + F + I,
tous_differents([A,B,C,D,E,F,G,H,I]).
```

3. Écrire une version optimisée du prédicat précédent, c'est-à-dire où les domaines de recherches sont réduits au maximum.

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

carre_magique(A,B,C,D,E,F,G,H,I,S) :-
between(0,100,S),
between(0,S,A),
between(0,S,B),
between(0,S,D),
between(0,S,E),
C is S - (A + B),
F is S - (D + E),
G is S - (A + D),
H is S - (B + E),
I is S - (G + H),
S =:= C + F + I,
tous_differents([A,B,C,D,E,F,G,H,I]).
```

4. On suppose que l'on a à notre disposition un prédicat `add(P,Q,R)` (resp. `mul(P,Q,R)`) qui est vrai si le polynôme `r` est la somme (resp. le produit) des polynômes `P` et `Q`. Écrire un prédicat `div(A,B,Q,R` vrai si `Q` et `R` sont les quotient et reste de la division de `A` par `B`.

```
div(A,B,Q,R} :- mul(B,Q,S), add(S,R,A).
```

12 TP5 : une histoire de (voisin du) fou.

TP noté

Ce TP sera à rendre pour le mercredi 26 mai au plus tard sous forme d'un fichier envoyé à : `vivien@icps.u-strasbg.fr`. Il vous est demandé de rendre un listing où vous commenterez en une ou deux phrases la réalisation de chacun des prédicats demandé. En SWI-Prolog, les commentaires se notent comme en C : `/* un commentaire pertinent */`.

En cas de comportement incompréhensible d'un prédicat, par exemple `long([1,2,3],X)`, n'hésitez pas à tracer son exécution : `trace, long([1,2,3],X)` (dans l'interpréteur).

Énoncé

Aux échecs, on veut faire parcourir toutes les cases de l'échiquier à un cavalier en s'arrangeant pour qu'il ne visite chaque case qu'une et une seule fois. On note N la taille de l'échiquier (qui est supposé être carré). On veut construire un prédicat qui nous permette de construire un tel parcours. Une case sera représentée par la liste de ses deux coordonnées.

1. Écrire un prédicat `suivant` qui prend en entrée N et les coordonnées de deux cases et qui est vrai si et seulement si un cavalier peut passer d'une case à l'autre. La figure 1 montre les mouvements possibles du cavalier. On supposera que les coordonnées de la première case sont bien dans les limites de l'échiquier.

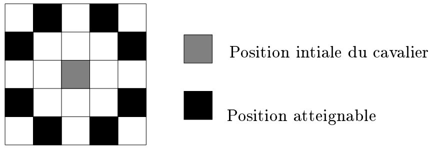


Fig. 1: Mouvements possibles d'un cavalier à partir d'une case donnée.

```
suivant(N,X,Y,U,V) :- U is X + 1, V is Y + 2, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X + 1, V is Y - 2, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X - 1, V is Y + 2, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X - 1, V is Y - 2, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X + 2, V is Y + 1, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X + 2, V is Y - 1, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X - 2, V is Y + 1, 1 =< U, U =< N, 1 =< V, V =< N.
suivant(N,X,Y,U,V) :- U is X - 2, V is Y - 1, 1 =< U, U =< N, 1 =< V, V =< N.
```

2. Écrire un prédicat `pas_elt` qui prend en entrée une case C et une liste L de cases et qui est vrai si et seulement si C n'appartient pas à L.

```
pas_elt(_,[]).
pas_elt([X,Y],[[U,_]|L]) :- X =\= U, !, pas_elt([X,Y],L).
pas_elt([X,Y],[[_,V]|L]) :- Y =\= V, !, pas_elt([X,Y],L).
```

3. Écrire un prédicat `balaye` qui prend en entrée N et deux listes de cases P et L. P représente la liste des cases déjà parcourues par le cavalier. `balaye` est vrai si et seulement si L prolonge P en un parcours de l'échiquier satisfaisant les conditions de l'énoncé.
Indication : relire les questions précédentes.

```
balaye(_,_,[]).
balaye(N,[[X,Y]|R],[[U,V]|S]) :- suivant(N,X,Y,U,V), pas_elt([U,V],[[X,Y]|R]),
          balaye(N,[[U,V],[X,Y]|R],S).
```

4. Écrire un prédicat qui prend en entrée une liste L et un entier M et qui est vrai si et seulement si M est la longueur de la liste L. Vérifier que ce prédicat permet de construire une liste d'une taille M donnée.

```
liste_size([],0).
liste_size([[_,_]|R],S) :- liste_size(R,U), S is U+1.
```

5. Écrire un prédicat `parcours` qui prenne en entrée un entier N, représentant la taille de l'échiquier, et une liste L, et qui permette de construire un parcours satisfaisant les conditions de l'énoncé et commençant par la première case de l'échiquier.
Indication pour le debuggage : le problème n'a pas de solutions pour des valeurs de N inférieures ou égales à 4, mais admet des solutions pour $N = 5$.

```
parcours(N,[[1,1]|L]) :- T is N*N-1, liste_size(L,T), !, balaye(N,[[1,1]],L).
```

13 TD10 : 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;;
```

```
val 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. Écrire une fonctionnelle qui réalise la composition de deux fontions.

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

3. Écrire une fonction qui calcule la somme des valeurs d'une fonction `f` sur les N premiers entiers. (Ne pas utiliser de boucle `for`.)

```
let rec somme f add neutre = function
   0 -> neutre
 | n -> add (somme f add neutre (n-1)) (f n);;
val somme : (int -> 'a) -> ('b -> 'a -> 'b) -> 'b -> int -> 'b = <fun>
```

4. Écrire, au moyen d'un itérateur, 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;;
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. Écrire, au moyen d'un itérateur, 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>
```

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).
```

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.
```