Annales des TD et TP de programmation fonctionnelle et logique

IUP 2 – année 1999-2000

1 TD 1 : définitions, filtrage

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>
```


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;;
 Characters 15-30:
Warning: this pattern-matching is not exhaustive.
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 paire 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 entré 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>
```

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

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 récursives simples sur les tableaux

1. Écrivez une fonction qui teste si deux tableaux sont égaux.


```
# let égalité u v =
   let lu = Array.length u and lv = Array.length v
   in let rec comparaison i =
      (i = lu) or ((u.(i) = v.(i)) & (comparaison (i+1)))
      in (lu = lv) & (comparaison 0);;
         val égalité : 'a array -> 'a array -> bool = <fun>
```

2. Écrivez une fonction qui calcule l'élément maximum d'un tableau d'entiers.

```
# let maximum tab =
   let length = Array.length tab
   in let rec maximum_aux i =
      if i = length-1
         then tab.(i)
         else max tab.(i) (maximum_aux (i+1))
      in match length with
           0 -> failwith "tableau vide !"
         | 1 -> tab.(0)
         | _ -> maximum_aux 0;;
                  val maximum : 'a array -> 'a = <fun>
```

Palindromes

Un palindrome est un mot qui se lit bien à l'endroit qu'à l'envers (cf. figure 1).

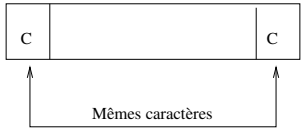

FIG. 1: Structure d'un palindrome

Rappels : l'expression `String.length s` renvoie la longueur de la chaîne de caractères `s` ; l'expression `String.sub s a b` construit une chaîne de `b` caractères égale à la sous-chaîne de `s` débutant par son `a`^{e}caractère.

– Écrire une fonction à valeurs booléennes qui est vrai si le mot qui lui est passé en argument est un palindrome.

  – La chaîne vide est un palindrome.
  – Un mot réduit à un seul caractère est un palindrome.
  – Un mot d'au moins deux caractères est un palindrome si :
    – ses premier et dernier caractères sont identiques ;
    – le mot situé entre ces caractères est un palindrome.

```
# let rec palindrome s =
   match String.length s with
     0 -> true
   | 1 -> true
   | n -> if s.[0]=s.[n-1]
          then palindrome (String.sub s 1 (n-2))
          else false;;
            val palindrome : string -> bool = <fun>
```

```
# let rec palindrome s =
   match String.length s with
     0 -> true
```

```
  | 1 -> true
  | n -> (s.[0]=s.[n-1]) & palindrome(String.sub s 1 (n-2));;
       val palindrome : string -> bool = <fun>
```

– Même question mais en ne construisant pas de sous-chaînes de caractères.
On utilise deux indices, `i` et `j`, désignant respectivement le début et la fin de la sous-chaîne traitée. Initialement, `i` vaut 0 et `j` la longueur de la chaîne moins 1. À chaque étape on compare les caractères d'indice `i` et `j` :
  – s'ils sont égaux on continue ;
  – s'ils sont différents on arrête ;
  – si $\texttt{i} \geq \texttt{j}$, on a atteint ou dépassé le milieu et on arrête.

```
# let rec palin s i j =
    (i >= j) or ((s.[i]=s.[j])&(palin s (i+1) (j-1)));;
   val palin : string -> int -> int -> bool = <fun>

# let palindrome s = palin s 0 (String.length s - 1);;
 val palindrome : string -> bool = <fun>

# let palindrome s =
    let rec palin i j =
      (i >= j) or ((s.[i]=s.[j])&(palin (i+1) (j-1)))
    in palin 0 (String.length s - 1);;
       val palindrome : string -> bool = <fun>
```

3 TD 3 : récursivité

Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

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

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

```
# 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 $n > 0$, $(\text{Fib}(n), \text{Fib}(n-1))$. L'utiliser pour écrire une fonction calculant $\text{Fib}(n)$.

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

Fonctions classiques sur les ensembles triés

On considère des ensembles triés représentés par des listes. Écrire des fonctions réalisant les opérations 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 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>
```

4. Différence symétrique ;

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

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

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

4 TD 4 : typage et polymorphisme

Expressions à typer

```
# 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;;
 Characters 22-23:
This expression has type bool but is here used with type int

# 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>
```

Addition polymorphe des éléments d'une liste

1. Écrivez une fonction récursive qui calcule la somme des éléments d'une liste d'entiers 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 et 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>
```

```
# 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 : 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)
```

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 fd : 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_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, 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 `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>
```

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
       (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>
```

7 TD 7 : logique

Analyse de vérité

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

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

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

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

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

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

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

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

$a=0$	$a=1$	
$((0 \vee b) \wedge c) \leftrightarrow ((0 \wedge c) \vee (b \wedge c))$ $(b \wedge c) \leftrightarrow (0 \vee (b \wedge c))$ $(b \wedge c) \leftrightarrow (b \wedge c)$ 1	$((1 \vee b) \wedge c) \leftrightarrow ((1 \wedge c) \vee (b \wedge c))$ $(1 \wedge c) \leftrightarrow (c \vee (b \wedge c))$ $c \leftrightarrow (c \vee (b \wedge c))$ $c \leftrightarrow (c \vee (b \wedge c))$	
	$b=0$	$b=1$
	$c \leftrightarrow (c \vee (1 \wedge c))$ $c \leftrightarrow (c \vee c)$ $c \leftrightarrow c$ 1	$c \leftrightarrow (c \vee (1 \wedge c))$ $c \leftrightarrow (c \vee c)$ $c \leftrightarrow c$ 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 subsection* précédente) :

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 :

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

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

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.

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

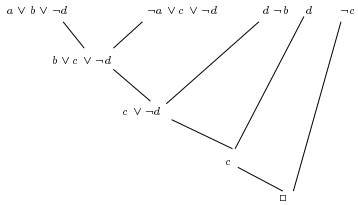

FIG. 2: Preuve de $((a \vee b \vee \neg d) \wedge (\neg a \vee c \vee \neg d) \wedge (\neg b) \wedge d) \to c$

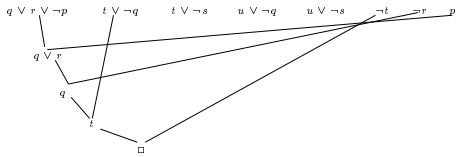

FIG. 3: Preuve de $(((p \to (q \vee r)) \wedge ((q \vee s) \to (t \wedge u))) \wedge \neg t) \to (p \to r)$

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 lettres 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.

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

La preuve de la formule : $((f \to g) \wedge ((g \wedge b) \to f) \wedge (b \to \neg f)) \leftrightarrow ((f \to g) \wedge (g \to \neg b))$ *est représentée par le tableau 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 bibliothèque.*

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.


```
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`.

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

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.

```
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).
```

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 ;

F = 9
M = 10989
```

10 TD 10 : révisions

Caml

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

```
#exception Erreur of int;;
 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. *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. (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 = <f
un>
```

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>
```

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.
```

11 TP 1 : listes et récursivité

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>
```

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 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>
```

Fonctions classiques sur les ensembles triés

Voir la section 3.

12 TP 2 : 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 = Feuille of int | Noeud of (int * arbre list);;
 type arbre = | Feuille of int | 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
          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
  | _ -> 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)
                                        0);;
        val hauteur : arbre -> int = <fun>

# let rec hauteur = function
     Feuille _   -> 1
  | 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
      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_feuilles = function
      Feuille _   -> 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
     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>

# let rec nb_noeuds = function
      Feuille _  -> 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
        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>

# let rec appartient n= function
    Feuille f
       -> f = 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>
```

3. 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, 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>
```

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
       (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);;
 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 -> arbre = <fun>
```

13 TP 3 : 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]).
```

14 TP 4 : le compte est bon

Ce TP est à rendre impérativement pour le vendredi 7 janvier, délai de rigueur. Vous devez m'envoyer votre programme par mail à l'adresse : `vivien@icps.u-strasbg.fr`. Vous pouvez réaliser ce TP seul ou en binôme.

Il s'agit ici, une fois de plus, de résoudre le problème du compte est bon. Nous ne nous intéresserons ici qu'aux solutions exactes (nous ne rechercherons donc pas la meilleure approximation).

Soit une formule permettant d'obtenir la valeur cible. La liste des entiers figurant dans cette formule est une permutation d'une sous-liste de la liste d'entiers arguments. Le principe du programme que vous allez devoir écrire est de fabriquer toutes les permutations de toutes les sous-listes possibles de la liste argument. Étant donnée une de ces permutations, les entiers apparaissant dans la formule à construire et l'ordre d'apparition de ces entiers sont fixés. Il ne reste plus qu'à combiner entre eux ces entiers au moyen des quatre opérations.

Conseil : pensez à tester vos prédicats au fur et à mesure.

1. Écrivez un prédicat `sousliste(A,B)` qui est vrai si `A` est une sous-liste de `B`.

```
sousliste([],[]).
sousliste(R,[_|S]) :- sousliste(R,S).
sousliste([T|R],[T|S]) :- sousliste(R,S).
```

2. (Ré-)écrivez un prédicat `permutation(A,B)` qui est vrai si la liste `B` est une permutation de la liste `A` dont la valeur est donnée en argument.

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

3. Écrivez un prédicat `décompose(A,B,C)` qui est vrai si et seulement si la liste `C` est la concaténation des listes `A` et `B` et si ni la liste `A`, ni la liste `B` n'est vide. Vous pouvez réutilisez le prédicat prédéfini `append` de concaténation des listes.

```
décompose([A|B],[C|D],E) :- append([A|B],[C|D],E).
```

4. Écrivez un prédicat `construit_chaîne(A,B,Op,C)` où `A` et `B` sont les chaînes de caractères représentants deux nombres à combiner par l'opération représentée par la chaîne de caractères `Op`. `C` est la chaîne de caractères résultante. Si `A = "3"`, `B="4"` et `Op="*"`, nous voulons obtenir `C="(3*4)"`.
   En SWI-Prolog les chaînes de caractères sont représentées par des tableaux d'entiers, comme l'illustre l'exemple suivant :

```
?- X = "bonjour".

X = [98, 111, 110, 106, 111, 117, 114]
```

```
Yes
```

Pour obtenir les chaînes de caractères sous une forme lisible, nous devons obligatoirement avoir recours aux fonctions ci-dessous :

```
?- string_to_list(X,"bonjour").

X = "bonjour"

Yes
```

```
?- string_to_atom(X,3).

X = "3"

Yes
```

```
?- string_to_list(X,"bon"),string_to_list(Y,"jour"),string_concat(X,Y,Z).

X = "bon"
Y = "jour"
Z = "bonjour" ;

No
```

```
construit_chaîne(A,B,Op,C) :-
   string_to_list(Paropen, "("),
   string_to_list(Parclose, ")"),
   string_concat(Paropen, A, Temp1),
   string_concat(Temp1, Op, Temp2),
   string_concat(Temp2, B, Temp3),
   string_concat(Temp3, Parclose, C).
```

5. Écrivez un prédicat `combine(A,B,C)` où `A`, `B` et `C` sont des listes à deux éléments : un nombre et la formule, sous forme de chaîne de caractères, permettant de l'obtenir. `A` et `B` sont les deux nombres à combiner et `C` est le résultat de cette combinaison via l'une des quatre opérations.

   Indication : on pourra utiliser l'opérateur `mod`.

```
combine([A,SA],[B,SB],[C,SC]) :-
   C is A+B,
   string_to_list(Plus,"+"),
   construit_chaîne(SA,SB,Plus,SC).

combine([A,SA],[B,SB],[C,SC]) :-
   A > B,
   C is A-B,
   string_to_list(Moins,"+"),
   construit_chaîne(SA,SB,Moins,SC).

combine([A,SA],[B,SB],[C,SC]) :-
   C is A*B,
   string_to_list(Mult,"*"),
   construit_chaîne(SA,SB,Mult,SC).

combine([A,SA],[B,SB],[C,SC]) :-
   A mod B =:= 0,
   C is A/B,
   string_to_list(Div,"/"),
   construit_chaîne(SA,SB,Div,SC).
```

6. Écrivez un prédicat `valeur(L,D)` où `L` est une liste d'entiers et `D` une liste à deux éléments : un entier et la formule, sous forme de chaîne de caractères, permettant de l'obtenir. Le prédicat `valeur(L,D)` doit construire à partir de `L` toutes les valeurs de `D`.

   Indication : ce prédicat décomposera, si possible, la liste `L` en deux sous-listes sur lesquelles le prédicat sera appelé récursivement.


```
valeur([],_) :- !, fail.
valeur([A],VA) :- !, string_to_atom(SA,A), VA=[A,SA].
valeur(L,D) :-
   décompose(X,Y,L), valeur(X,VX), valeur(Y,VY), combine(VX,VY,D).
```

7. Combinez les prédicats précédemment écrits pour obtenir un prédicat `lecompteestbon(L,C,D)` qui, étant donnés un entier cible `C` et une liste d'entiers `L`, construit, si possible, une liste à deux éléments constituée de l'entier cible et d'une formule, sous forme de chaîne de caractères, permettant de l'obtenir à partir des éléments de la liste `L`.

```
réussite(C,[A,_]) :- C = A.

lecompteestbon(L,C,D) :- sousliste(M,L), permutation(M,N),
valeur(N,D), réussite(C,D),!.
```

15 Examen de PFEL de juin 1999

Aucun document n'était autorisé. Le sujet n'était pas prévu pour être fait en trois heures !

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 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>
```

Exercice 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) ...))`.

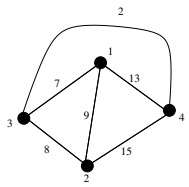

FIG. 4: Exemple de problème avec quatre villes.

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>
```

Exercice 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 4.)

– 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 4 : $[Chemin(9,[1;2]); Chemin(8,[2;3]); Chemin(7,[1;3]); Chemin(13,[4;1]); Chemin(15,[4;2]); Chemin(2,[4;3])]$.

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 `tousleschemins` 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>
```

Exercice 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 4.)
- 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 4.)

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.

16 Projet : le compte est bon

16.1 Présentation du sujet

Le problème est le suivant : nous disposons d'un ensemble d'entiers naturels (les valeurs de départ) et d'un entier cible n, et nous voulons combiner les valeurs de départ au moyen des quatre opérations de manière à obtenir l'entier p le plus proche possible de n, sachant que nous pouvons utiliser au plus une fois chacune des valeurs de départ. Nous sommes intéressés non seulement par la valeur de p mais aussi par la formule qui nous a permis de l'obtenir. De ce fait, l'objet de base que vous manipulerez sera une structure combinant une expression mathématique et sa valeur. Cette structure est définie comme suit :

```
# type nombre={nb: int; expr: string};;
 type nombre = { nb: int; expr: string }
# let a={nb=782; expr="(((5*7)-1)*(25-2))"};;
 val a : nombre = {nb=782; expr="(((5*7)-1)*(25-2))"}
# let valeur = function {nb = n ; expr = _} -> n;;
 val valeur : nombre -> int = <fun>
```

Dans un premier temps, vous chercherez simplement à construire à partir des valeurs de départ toutes les expressions possibles (subsection 16.2). Ensuite vous modifierez vos fonctions pour qu'elles renvoient une expression permettant d'obtenir l'entier cible, si une telle expression figure parmi celles que vous aurez construites (subsection 16.3). Finalement, vous modifierez encore vos fonctions pour qu'elles renvoient la meilleure approximation possible (subsection 16.4).

Chaque fonction demandée est décrite aussi précisément que possible et son type est précisé. En plus, un exemple d'utilisation est fourni à chaque fois. Ces deux descriptions, littérale et par l'exemple, sont volontairement redondantes pour faciliter la compréhension du sujet.

Fonctions que vous pourriez ne pas connaître mais dont vous pourriez avoir besoin : `mod` (modulo), `@` (concaténation de listes), `^` (concaténation de chaînes de caractères) et `string_of_int`.

Travail demandé

Ce travail est à effectuer seul ou en binôme, à l'exclusion de tout regroupement de taille supérieure. Pour le vendredi 17 décembre vous devrez m'envoyer par mail (adresse : vivien@icps.u-strasbg.fr) le code de *toutes* vos fonctions, c'est-à-dire trois fichiers, l'un contenant les fonctions de la subsection 16.2, l'autre celles de la subsection 16.3 et le troisième celles de la subsection 16.4. Pour la même date vous me rendrez un rapport sommaire (une page) m'expliquant les points délicats du programme.

La taille de l'ensemble des formules possibles croit très rapidement avec le nombre des valeurs de départ. Mon propre programme n'est pas toujours capable de traiter des ensembles de six valeurs de départ.

16.2 Construction de toutes les combinaisons possibles

1. Nous avons besoin d'une fonction `combinaison` qui, étant donnée une liste `l`, nous construise, sous la forme d'une liste de triplets, toutes les combinaisons possibles formées de deux éléments de la liste `l` et de la liste des autres éléments de `l`.

```
# let combinaison l =
  let rec extrait_deuxième premier l accu =
    match l with
      t::[] -> [(premier,t,accu)]
    | t::r  -> (premier,t,accu@r)::(extrait_deuxième premier r (t::accu))
    | _   -> failwith "Liste vide in extrait_deuxième"
  in let rec extrait_premier l accu =
    match l with
      a::b::[] -> [(a,b,accu)]
    | a::b::r  -> (extrait_deuxième a (b::r) (accu))
                  @(extrait_premier (b::r) (a::accu))
    | _   -> failwith "Liste de mois d'un élément in extrait_premier"
  in extrait_premier l [];;
                          val combinaison : 'a list -> ('a * 'a * 'a list
) list = <fun>

# combinaison;;
 - : 'a list -> ('a * 'a * 'a list) list = <fun>
# combinaison [1;2;3];;
 - : (int * int * int list) list = [1, 2, [3]; 1, 3, [2]; 2, 3, [1]]
```

Vous remarquerez que dans cet exemple figure l'élément `(1, 2, [3])` mais pas le triplet `(2, 1, [3])`, l'ordre relatif des deux premiers éléments du triplet ne nous important pas.

2. Nous voulons maintenant pouvoir combiner deux nombres au moyen d'une des quatre opérations. Vous écrirez donc quatre fonctions (`addition`, `soustraction`, `multiplication` et `division`) chacune prenant en entrée un triplet formé de deux objets de type `nombre` et d'une liste d'éléments de type `nombre`, et retournant une liste contenant une liste de `nombre`, cette dernière liste contenant la valeur calculée à partir des deux premiers éléments du triplet ainsi que le contenu du troisième élément du triplet.

```
# let addition n =
    match n with
      ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
        [{nb = va+vb; expr = "("^fa^"+"^fb^")"}::l];;
        val addition : nombre * nombre * nombre list -> nombre list list
= <fun>

# let soustraction n =
    match n with
      ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
        if va <> vb
        then if va > vb
             then [{nb = va-vb; expr = "("^fa^"-"^fb^")"}::l]
```

```
            else [{nb = vb-va; expr = "("^fb^"-"^fa^")"}::l]
       else [];;
               val soustraction : nombre * nombre * nombre list -> nombr
e list list = <fun>

# let multiplication n =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       if (va <> 1) & (vb <> 1)
           then [{nb = va*vb; expr = "("^fa^"*"^fb^")"}::l]
           else [];;
           val multiplication : nombre * nombre * nombre list -> nombre
list list =
  <fun>

# let division n =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       if (va <> 1) & (vb <> 1)
       then if (va mod vb) = 0
            then [{nb = va/vb; expr = "("^fa^"/"^fb^")"}::l]
            else if (vb mod va) = 0
                 then [{nb = vb/va; expr = "("^fb^"/"^fa^")"}::l]
                 else []
       else [];;
                  val division : nombre * nombre * nombre list -> nombr
e list list = <fun>

# addition;;
 - : nombre * nombre * nombre list -> nombre list list = <fun>

# addition ({nb=3; expr="(2+1)"}, {nb=20; expr="(4*5)"},
          [{nb=7; expr="7"}; {nb=2; expr="(6/3)"}]);;
   - : nombre list list =
[[{nb=23; expr="((2+1)+(4*5))"}; {nb=7; expr="7"}; {nb=2; expr="(6/3)"}]
]
```

Remarques. Vous ferez attention à ne pas générer des expressions inutiles (multiplication par un). Vous veillerez aussi à n'effectuer que des divisions « qui tombent juste » et à ne pas générer de valeurs négatives.

Question. Pourquoi le « prof » a-t-il pu juger préférable que ces fonctions renvoient des objets de type `nombre list list` plutôt que des objets de type `nombre list` ?

3. Nous voulons ici une fonction `opérations` qui, étant donné un triplet comme celui utilisé par les fonctions de la question précédente, construit la liste des résultats de l'application des différentes opérations à cette argument.

```
# let opérations n =
    (addition n)@(soustraction n)@(multiplication n)@(division n);;
  val opérations : nombre * nombre * nombre list -> nombre list list =
<fun>

# opérations;;
 - : nombre * nombre * nombre list -> nombre list list = <fun>

# opérations ({nb=3; expr="3"}, {nb=6; expr="6"}, [{nb=7; expr="7"}]);;
 - : nombre list list =
[[{nb=9; expr="(3+6)"}; {nb=7; expr="7"}];
 [{nb=3; expr="(6-3)"}; {nb=7; expr="7"}];
 [{nb=18; expr="(3*6)"}; {nb=7; expr="7"}];
 [{nb=2; expr="(6/3)"}; {nb=7; expr="7"}]]
```


4. Nous voulons maintenant, partant d'une liste `l` de nombres, construire de nouvelles listes de nombres, chacune de ces nouvelles listes contenant la combinaison de deux éléments de `l` ainsi que les autres éléments de `l`.

```
# let combine_plaques l =
   List.fold_right (function x-> function y-> (opérations x)@y)
                   (combinaison l)
                   [];;
       val combine_plaques : nombre list -> nombre list list = <fun>

# combine_plaques;;
 - : nombre list -> nombre list list = <fun>

# combine_plaques [{nb=3; expr="3"}; {nb=6; expr="6"}; {nb=7; expr="7"}];;
 - : nombre list list =
[[{nb=9; expr="(3+6)"}; {nb=7; expr="7"}];
 [{nb=3; expr="(6-3)"}; {nb=7; expr="7"}];
 [{nb=18; expr="(3*6)"}; {nb=7; expr="7"}];
 [{nb=2; expr="(6/3)"}; {nb=7; expr="7"}];
 [{nb=10; expr="(3+7)"}; {nb=6; expr="6"}];
 [{nb=4; expr="(7-3)"}; {nb=6; expr="6"}];
 [{nb=21; expr="(3*7)"}; {nb=6; expr="6"}];
 [{nb=13; expr="(6+7)"}; {nb=3; expr="3"}];
 [{nb=1; expr="(7-6)"}; {nb=3; expr="3"}];
 [{nb=42; expr="(6*7)"}; {nb=3; expr="3"}]]
```

Cette fonction doit être réalisée au moyen d'une fonctionnelle.

5. Nous nous plaçons maintenant à un niveau plus élevé : nous avons une liste de liste de nombres, et à chaque élément de notre liste nous voulons appliquer le traitement réalisé par la fonction précédente.

```
# let nouvelles_valeurs l =
   List.fold_right (function x -> function y -> (combine_plaques x)@y)
                   l
                   [];;
       val nouvelles_valeurs : nombre list list -> nombre list list = <f
un>

# nouvelles_valeurs;;
 - : nombre list list -> nombre list list = <fun>

# nouvelles_valeurs [[{nb=1; expr="(7-6)"}; {nb=3; expr="3"}];
  [{nb=42; expr="(6*7)"}; {nb=3; expr="3"}]];;
   - : nombre list list =
[[{nb=4; expr="((7-6)+3)"}]; [{nb=2; expr="(3-(7-6))"}];
 [{nb=45; expr="((6*7)+3)"}]; [{nb=39; expr="((6*7)-3)"}];
 [{nb=126; expr="((6*7)*3)"}]; [{nb=14; expr="((6*7)/3)"}]]
```

Cette fonction doit être réalisée au moyen d'une fonctionnelle.

6. Finalement, écrivez une fonction `toutes_les_combinaisons` qui prend en entrée une liste d'entiers et qui fabrique successivement toutes les combinaisons possibles de ces entiers. Cette fonction ne retournera pas la liste de toutes les combinaisons possibles, mais tous les nombres que l'on peut obtenir en combinant *tous* les entiers de la liste argument.

```
# let rec initiation = function
      [] -> []
    | t::r -> {nb = t; expr = (string_of_int t)}::(initiation r);;
      val initiation : int list -> nombre list = <fun>

#
 let toutes_les_combinaisons nombres =
   let rec itérations l = function
```

```
      0 -> failwith "Pas de nombres à combiner"
    | 1 -> l
    | n -> itérations (nouvelles_valeurs l) (n-1)
  in let liste_nombres = initiation nombres
     in itérations [liste_nombres] (List.length liste_nombres);;
            val toutes_les_combinaisons : int list -> nombre list lis
t = <fun>

#toutes_les_combinaisons;;
 - : int list -> nombre list list = <fun>

#toutes_les_combinaisons [1;2;3];;
 - : nombre list list =
[[{nb=6; expr="((1+2)+3)"}]; [{nb=9; expr="((1+2)*3)"}];
 [{nb=1; expr="((1+2)/3)"}]; [{nb=4; expr="((2-1)+3)"}];
 [{nb=2; expr="(3-(2-1))"}]; [{nb=6; expr="((1+3)+2)"}];
 [{nb=2; expr="((1+3)-2)"}]; [{nb=8; expr="((1+3)*2)"}];
 [{nb=2; expr="((1+3)/2)"}]; [{nb=4; expr="((3-1)+2)"}];
 [{nb=4; expr="((3-1)*2)"}]; [{nb=1; expr="((3-1)/2)"}];
 [{nb=6; expr="((2+3)+1)"}]; [{nb=4; expr="((2+3)-1)"}];
 [{nb=2; expr="((3-2)+1)"}]; [{nb=7; expr="((2*3)+1)"}];
 [{nb=5; expr="((2*3)-1)"}]]
```

16.3 Formule s'évaluant en le nombre recherché

Le but du jeu, ici, est d'arrêter les constructions de nouvelles valeurs dès qu'une formule permettant de trouver le nombre recherché a été trouvée, et de retourner alors cette formule.

1. Réécrivez chacune des fonctions `addition`, `soustraction`, `multiplication` et `division` pour qu'elle compare au nombre cible la formule qu'elle construit. Quel doit être le comportement de la fonction en cas d'égalité ?

```
#exception Trouvé of nombre;;
 exception Trouvé of nombre

#let test cible l ({nb = v; expr = f}as t)=
   if v=cible
   then raise (Trouvé t)
   else [t::l];;
       val test : int -> nombre list -> nombre -> nombre list list = <fu
n>

#let addition n cible =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       test cible l {nb = va+vb; expr = "("^fa^"+"^fb^")"};;
       val addition : nombre * nombre * nombre list -> int -> nombre lis
t list =
  <fun>

#let soustraction n cible =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       if va <> vb
       then if va > vb
            then test cible l
                      {nb = va-vb; expr = "("^fa^"-"^fb^")"}
            else test cible l
                      {nb = vb-va; expr = "("^fb^"-"^fa^")"}
       else [];;
```



```
                  val soustraction : nombre * nombre * nombre list -> i
nt -> nombre list list =
  <fun>

#let multiplication n cible =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       if (va <> 1) & (vb <> 1)
           then test cible l {nb = va*vb; expr = "("^fa^"*"^fb^")"}
           else [];;
           val multiplication :
  nombre * nombre * nombre list -> int -> nombre list list = <fun>

#let division n cible =
   match n with
     ({nb = va; expr = fa},{nb = vb; expr = fb},l) ->
       if (va <> 1) & (vb <> 1)
       then if (va mod vb) = 0
            then test cible l {nb = va/vb; expr = "("^fa^"/"^fb^")"}
            else if (vb mod va) = 0
                 then test cible l {nb = vb/va; expr = "("^fb^"/"^fa^")"}
                 else []
       else [];;
                  val division : nombre * nombre * nombre list -> int -
> nombre list list =
  <fun>

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

2. Modifiez les autres fonctions pour que l'ensemble de vos fonctions puisse avoir le comportement requis. Il faudra, entre autres, que la fonction `toutes_les_combinaisons` prenne l'entier cible en argument.

```
#let opérations n cible =
   (addition n cible)@(soustraction n cible)@
   (multiplication n cible)@(division n cible);;
     val opérations : nombre * nombre * nombre list -> int -> nombre lis
t list =
  <fun>

#let combine_plaques l cible =
   List.fold_right (function x-> function y-> (opérations x cible)@y)
                   (combinaison l)
                   [];;
       val combine_plaques : nombre list -> int -> nombre list list = <f
un>

#let nouvelles_valeurs l cible =
   List.fold_right (function x -> function y -> (combine_plaques x cible)@y)
                   l
                   [];;
       val nouvelles_valeurs : nombre list list -> int -> nombre list li
st = <fun>

#let rec initiation = function
     [] -> []
   | t::r -> {nb = t; expr = (string_of_int t)}::(initiation r);;
     val initiation : int list -> nombre list = <fun>

#let toutes_les_combinaisons nombres cible =
   let rec itérations l = function
       0 -> failwith "Pas de nombres à combiner"
```

```
      | 1 -> l
      | n -> itérations (nouvelles_valeurs l cible) (n-1)
   in let liste_nombres = initiation nombres
      in try itérations [liste_nombres] (List.length liste_nombres)
         with Trouvé n -> [[n]];;
              val toutes_les_combinaisons : int list -> int -> nombre l
ist list = <fun>

#toutes_les_combinaisons [2;3] 7;;
 - : nombre list list =
[[{nb=5; expr="(2+3)"}]; [{nb=1; expr="(3-2)"}]; [{nb=6; expr="(2*3)"}]]


#toutes_les_combinaisons [2;3] 5;;
 - : nombre list list = [[{nb=5; expr="(2+3)"}]]
```

16.4 Meilleure approximation

Nous voulons finalement un ensemble de fonctions qui nous retourne, parmi les formules que l'on veut construire, celle dont la valeur est la plus proche de l'entier cible.

1. Tout d'abord, nous voulons une fonction qui, étant donnés l'entier cible et deux objets de type `nombre`, nous renvoie celui dont la valeur est la plus proche de celle de l'entier cible.

```
#let meilleure cible approx_1 approx_2 =
   match (approx_1, approx_2) with
     ({nb = v_1; expr = _ }, {nb = v_2; expr = _ }) ->
       if abs(v_1 - cible) < abs (v_2 - cible)
           then approx_1
           else approx_2;;
           val meilleure : int -> nombre -> nombre -> nombre = <fun>

#meilleure;;
 - : int -> nombre -> nombre -> nombre = <fun>

#meilleure 7 {nb=4;expr="4"} {nb=9;expr="9"};;
 - : nombre = {nb=9; expr="9"}
```

2. Les valeurs que nous créons au moyen des fonctions `addition`, `soustraction`, `multiplication` et `division` sont immédiatement stockées en tête d'une liste de nombres. Nous voulons donc une fonction qui, étant donnés l'entier cible, une approximation `n` et une liste `l` de nombres comme celle décrite ci-dessus, nous renvoie la meilleure approximation de la tête de liste `l` et de `n`.

```
#let plus_proche cible approx l =
   match l with
     t::r -> meilleure cible approx t
   | _ -> failwith "Liste vide in plus_proche ?";;
      val plus_proche : int -> nombre -> nombre list -> nombre = <fun>


#plus_proche;;
 - : int -> nombre -> nombre list -> nombre = <fun>

#plus_proche 8 {nb=9;expr="9"} [{nb=4;expr="4"} ; {nb=8;expr="8"}];;
 - : nombre = {nb=9; expr="9"}
```

3. Nous voulons maintenant une fonction qui nous renvoie la meilleure approximation de l'entier cible entre une approximation donnée en argument et celles qui sont contenues dans une liste de liste de nombres, en ne tenant compte, comme dans la question précédente, que du premier élément de chaque liste de nombres.


```
#let meilleure_approx cible approx l =
   List.fold_left (plus_proche cible) approx l;;
   val meilleure_approx : int -> nombre -> nombre list list -> nombre =
<fun>

#meilleure_approx;;
 - : int -> nombre -> nombre list list -> nombre = <fun>

#meilleure_approx 7 {nb=9;expr="9"}
       [[{nb=4;expr="4"} ; {nb=8;expr="8"}]; [{nb=6;expr="6"};{nb=8;expr="8"}]];;
   - : nombre = {nb=6; expr="6"}
```

Cette fonction doit être réalisée au moyen d'une fonctionnelle.

4. Finalement, écrivez une fonction `la_meilleure_combinaison` qui résolve notre problème.

```
#let init_approx cible l =
   match l with
     t::r -> List.fold_left (meilleure cible) t r
   | _   -> failwith "Liste vide in init_approx";;
      val init_approx : int -> nombre list -> nombre = <fun>

#let la_meilleure_combinaison nombres cible =
   let rec itérations l approx = function
       0 -> failwith "Pas de nombres à combiner"
     | 1 -> approx
     | n -> let nouvelle_liste = nouvelles_valeurs l cible
            in let nouvelle_approx = meilleure_approx cible approx nouvelle_liste
               in itérations nouvelle_liste nouvelle_approx (n-1)
   in let liste_nombres = initiation nombres
      in let approx = init_approx cible liste_nombres
         in try itérations [liste_nombres] approx (List.length liste_nombres)
            with Trouvé n -> n;;
                    val la_meilleure_combinaison : int list -> int -> n
ombre = <fun>

#la_meilleure_combinaison [2;4;3;7;25] 1999;;
 - : nombre = {nb=2000; expr="((4*(3+7))*(2*25))"}

#la_meilleure_combinaison [2;4;3;7;25] 2000;;
 - : nombre = {nb=2000; expr="((4*(3+7))*(2*25))"}
```