Programmation fonctionnelle et logique
Corrigé du projet : le compte est bon

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 (section 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 (section 3). Finalement, vous modifierez encore vos fonctions pour qu'elles renvoient la meilleure approximation possible (section 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 section 2, l'autre celles de la section 3 et le troisième celles de la section 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.

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)"}]]
```

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)"}]]
```

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))"}
```