# TP 4: Fractions Continues et Compte est bon

**NB**: pour toutes vos fonctions vous devez bien préciser leurs types !

## 1 Fractions Continues

Pour cet exercice vous devrez implémenter un type de Haskell qui représente les fractions continues. Une fraction continue est une méthode d'approximer n'importe quel réel positif avec une expression qui a la forme suivante :

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{a_5 + \dots}}}}}$$

où $a_0, a_1, \dots$ sont des entier positifs. L'intérêt de cette méthode est que souvent elle a une convergence plus rapide que la représentation classique (décimale) des nombres réels.

Pour plus d'information sur les fraction continues, vous pouvez regardez l'article correspondant de Wikipedia `https://fr.wikipedia.org/wiki/Fraction_continue`.

### 1.1 Implémentation Haskell

**NB**: Pour cet exercice on va utiliser (entre autres) le type `Rational` qui représente les nombres rationnels. Il faudra donc ajouter au début de votre programme la ligne `import GHC.Real`. Pour rappel, les rationnels sont construits avec l'opération `%`. Par exemple :

```
*Main> (3%5) + (2%3)
19 % 15
```

Nous revenons aux fractions continues. Notre première observation est que les fractions continues ont une forme récursive : le dénominateur de chaque fraction est aussi une fraction continue. Cela nous motive de définir un type de Haskell comme suit :

```
data Frac = Frac Integer (Maybe Frac)
```

Votre objectif pour cette exercice est de fournir des fonctions qui vont permettre à l'utilisateur de se servir de ce type pour approximer des nombres réels (`Double`) et généralement pour manipuler des fractions continues. Vous devez implémenter (au moins) les fonctions suivantes :

1. Une fonction `list2frac :: [Integer] -> Frac` qui étant donné une liste d'entiers $[a_0, a_1, \dots, a_n]$ retourne la fraction continue qui correspond.
2. Une fonction `evalFrac :: Frac -> Rational` qui retourne un `Rational` qui correspond à la fraction continue donnée.

3. Une fonction `double2frac :: Int -> Double -> Double -> Frac` qui, étant donné un degré de précision (un `Int` qui représente la profondeur de la fraction continue et un double qui représente une erreur acceptable) et un `Double` $x$, retourne la fraction continue de cette précision qui approche $x$.

4. Faire en sorte que le type `Frac` fasse partie de la classe `Show` et les fractions continues sont bien affichées. "Bien affichées" veut dire, de manière lisible, sur plusieurs lignes, bien alignées. (Voir exemples ci-dessous)

## 1.2 Comment faire

Pour la fonction `double2frac` vous allez écrire une fonction qui prend trois arguments : la profondeur maximum `d`, l'erreur `eps`, et la valeur cible `x`. Vous devrez utilisez la formulation récursive suivante : on cherche la fraction continue qui approche le nombre réel $x$.

1. On met $a_0 = \lfloor x \rfloor$
2. On a $x' = x - a_0 < 1$. Si $x' <$`eps` on arrête, car le reste est suffisamment petit pour être ignoré.
3. Sinon, on définit $y = 1/x'$, qui est un nombre réel $> 1$.
4. On appelle récursivement la même fonction pour approcher $y$. Ça retourne une fraction continue $f$.
5. On met $x \approx a_0 + \frac{1}{f}$.

Évidemment, la quatrième étape doit diminuer la profondeur de l'approximation. Les fonctions suivantes vous seront peut-être utiles : `recip` calcule l'inverse de son argument, `fromInteger` convertit un `Integer` en `Double`.

Pour l'affichage sur plusieurs lignes vous devrez bien sûr implémenter la fonction `show`. Dans la chaîne de caractères que vous retournerez vous pouvez utiliser `'\n'` qui a le même sens qu'en C.

## 1.3 Exemples d'utilisation

**NB:** Votre implémentation devra fonctionner correctement pour tous les exemples suivants.

```
*Main> list2frac [4..8]
          1
4 + -------------
            1
    5 + ---------
              1
        6 + -----
                1
            7 + -
                8
*Main> double2frac 10 0.0001 (0.23)
          1
0 + -------------
            1
    4 + ---------
              1
        2 + -----
                1
            1 + -
                7
```

```
*Main> evalFrac $ list2frac $ take 30 [1,1..] -- Nombre d'or
1346269 % 832040
*Main> 1346269 / 832040
1.6180339887505408
*Main> double2frac 6 0.0001 (sqrt 2)
              1
1 + ---------------------
                1
    2 + -----------------
                  1
        2 + -------------
                    1
            2 + ---------
                      1
                2 + -----
                        1
                    2 + -
                        2
```

**Quelques explications** Deux questions se posent naturellement : pourquoi on vous demande que `evalFrac` retourne un `Rational` et pas simplement un `Double` et pourquoi on a besoin de l'argument `eps` dans la fonction `double2frac` ? La réponse est que les doubles en Haskell (comme en C et la plupart de langages) ont une précision finie, qui peut très facilement être la cause de plusieurs erreurs d'arrondi. Par exemple, considérez la ligne suivante :

```
Prelude> recip (recip 0.75 - 1)
3.000000000000001
```

Dans notre programme, ce type de problème peut se manifester comme suit :

```
double2frac 10 0.0000000000001 (0.23)
```

Théoriquement, cela devrait retourner exactement le même résultat que l'exemple qu'on a donné ci-dessus, car la fraction continue atteint la valeur $0,23$ exactement. Or, cela donne :

```
*Main> double2frac 10 0.0000000000001 (0.23)
                      1
0 + ---------------------------------
                        1
    4 + -------------------------------
                          1
        2 + -------------------------
                            1
            1 + ---------------------
                              1
                7 + ------------------
                                   1
                    8935713546370 + --
                                  32
```

La quantité $\frac{1}{8935713546370}$ est due à une erreur d'arrondi. Pour éviter qu'on finisse avec de telles erreurs dans notre fraction continue, on donne donc une valeur `eps` telle que toute valeur inférieure à `eps` peut être traitée comme insignifiante. Par contre, quand on convertit une fraction continue en `Rational` on évite ce type de problème, car les rationnels sont constitués de deux `Integer` et ont donc une précision (théoriquement) infinie.

# 2 Le compte est bon

Pour cet exercice on va programmer une variation du jeu "Le Compte est Bon". Pour rappel, dans ce jeu on nous donne une liste d'entiers et un entier cible. L'objectif est d'utiliser les entiers de la liste, au plus une fois chacun, et faire un calcul qui arrive à l'entier cible en utilisant les opérations arithmétiques de base.

L'objectif de cet exercice est de programmer en Haskell une fonction qui peut résoudre des problèmes de ce type, mais avec les variations suivantes :

- L'expression trouvée doit utiliser tous les entiers de la liste exactement une fois chacun.
- On permet seulement les opérations suivantes : addition, soustraction, multiplication. (Donc, pas de divisions, et pas de problèmes avec de division par zéro.)

## 2.1 Spécifications

Vous devez utiliser le type de Haskell suivant :

```
data Expr = Num Integer | Plus Expr Expr | Mult Expr Expr | Sub Expr Expr
```

L'idée est qu'une expression arithmétique est soit un entier, soit une addition/multiplication/soustraction de deux expressions. Programmez (au moins) les fonctions suivantes :

1. `show` et faire en sort que `Expr` soit membre de la classe `Show`.
2. `eval::Expr->Integer` qui retourne le résultat d'une expression.
3. `comptebon :: [Integer] -> Integer -> Maybe Expr` qui retourne une expression qui utilise tous les éléments du premier argument une fois chacun et a comme résultat le deuxième argument. Notez bien que le type de retour est `Maybe Expr` (et pas `Expr`) car c'est possible qu'une telle expression n'existe pas. Dans ce cas, votre fonction doit retourner `Nothing`.

## 2.2 Comment faire

La structure récursive du type `Expr` est importante à utiliser. Ainsi, chaque expression qui peut être construite avec la liste donnée est décomposée en deux sous-expressions. Les éléments utilisées par chaque sous-expression donnent une partition de la liste.

On vous demande de donner un algorithme de recherche brute–force qui, étant donné la liste d'entiers, produit toutes les expressions réalisables à partir de cette liste et vérifie s'il existe une parmi elles pour laquelle "le compte est bon". Donc, l'efficacité n'est pas une priorité pour cet exercice. Or, votre programme devrait fonctionner raisonnablement pour les exemples suivants. Quand il y plusieurs solutions possibles, vous pouvez retourner n'importe laquelle.

## 2.3 Exemples

```
*Main> comptebon [1,2,3,4,5] 50
Just ((((5-1)*3)*4)+2)
*Main> comptebon [1,2,3,4] 50
Nothing
*Main> comptebon [1,2,3,4,5,6] 50
Just ((((1+3)*2)*5)+4)+6)
*Main> comptebon [1,12,23,45,52] 512
Just (((1-52)+(12*45))+23)
```