TD 3 : Types

1 Types → Expressions

Pour chacun des types suivants, donnez une expression de Haskell qui lui correspond.

1. `[ (Int, Float) ]`
2. `[Int -> Int]`
3. `a -> (a,a)`
4. `(a,a) -> a`
5. `(Int->Int)->(Int->Int)->Int->Int`
6. `a->[a]->[a]`
7. `a->[b]`
8. `a -> b`

NB : Vous allez constater que vérifier vos solutions n'est pas forcément simple. Par exemple, si vous avez pensé à l'expression `[ (2,3.4) ]` pour la première question, vous pouvez essayer de faire `:t [(1,3.4)]` dans ghci. Or, cela retourne, `[(1,3.4)] :: (Num t1, Fractional t) => [(t1, t)]`, car ghci donne le type **le plus générique** qui correspond à l'expression. Cela n'empêche que l'expression corresponde aussi à un type plus spécifique. Donc, pour vérifier avec ghci que votre expression correspond bien au type demandé vous pouvez préciser le type explicitement avec l'opérateur `::`. Par exemple, on écrit

```
*Main>  [(1,3.4)] :: [(Int,Float)]
[(1,3.4)]
*Main>  [(1,3.4)] :: [(Int,Integer)]
<interactive>:48:6: error:...
```

Solution :

```
x1 :: [ (Int, Float) ]
x1 = [ (2,5.3) ]

x2 :: [Int ->Int]
x2 = (+) 2 : []

x3 :: a -> (a,a)
x3 x = (x,x)

x4 :: (a,a)->a
x4 (x,y) = x --attn NOT x4 (x,x)!!
ou
x4 (x,y) = head [x,y]

x5 ::  (Int->Int)->(Int->Int)->Int->Int
x5 f g x = f (g x)
```

```
x6 :: a -> [a] -> [a]
x6 = (:)

x7 :: a -> [b]
x7 _ = []

x8 :: a -> b
x8 x = x8 x
```

2 Expressions → Types

Pour les expressions suivantes (`y1,..,y7`), quel est le type inféré par ghci (si les expressions sont bien typées) ?

1. `y1 = 5.`
2. `y2 x = x+5`
3. `y3 f g = f (head g) : tail g`
4. `y4 x = [x]`
5. `y5 f 0 x = x`
   `y5 f n x = f (y5 f (n-1) x)`
6. `y6 f 0 = []`
   `y6 f 1 = [f]`
   `y6 f n = (f):(y6 f (n-1))`
7. `y7 x y z = if x then y else z`
8. `y6 y2 3`
9. `y3 (y6 y2 3)`
10. `y5 y4 2 "a"`

Solution :

```
*Main> :t y1
y1 :: Integer
*Main> :t y2
y2 :: Num a => a -> a
*Main> :t y3
y3 :: (a -> a) -> [a] -> [a]
*Main> :t y4
y4 :: t -> [t]
*Main> :t y5
y5 :: (Eq t, Num t) => (t1 -> t1) -> t -> t1 -> t1
*Main> :t y6
y6 :: (Eq t, Num t) => t1 -> t -> [t1]
*Main> :t y7
y7 :: Bool -> t -> t -> t
*Main> :t y6 y2 3
y6 y2 3 :: Num a => [a -> a]
*Main> :t y3 (y6 y2 3)
<interactive>:1:5: error:..
*Main> :t y5 y4 2 "a"
<interactive>:1:4: error:...
```

Pour corriger `y3 (y6 y2 3)` on constate que l'argument est une liste de fonctions (et pas une fonction). On aurait pu écrire

```
*Main> :t y3 (head (y6 y2 3))
y3 (head (y6 y2 3)) :: Num a => [a] -> [a]
```

Le problème de `y5 y4 2 "a"` est plus compliqué. On pourrait s'attendre à une expression qui donne `[["a"]]`. Or, le premier argument de `y5` a type `t1->t1`, donc la fonction `y4` ne correspond pas. Implicitement, le problème ici est que `y5` doit bien avoir un type qui ne dépend pas de la valeur de `n`.

3 Répétitions

Écrire une fonction qui, étant donné une fonction f et deux entiers n, x retourne le résultat d'appliquer la fonction f sur x, n fois. (Donc, $f(f(f(f \dots f(x) \dots)$.

Utiliser cette fonction pour donner une fonction qui implémente la multiplication (en utilisant l'opération `(+)` en répétition) et une fonction qui calcule la puissance x^y, pour x, y entier, en utilisant votre implémentation de la multiplication.

Solution :

```
applyntimes :: (Int->Int)->Int->Int->Int
applyntimes f 0 x = x
applyntimes f n x = f (applyntimes f (n-1) x)

multiplication :: Int -> Int -> Int
multiplication x y = applyntimes (+ x) y 0

power :: Int -> Int -> Int
power x y = applyntimes (multiplication x) y 1
```

4 Count Frequency

Écrire une fonction qui étant donné une liste `xs` et un élément `x` retourne le nombre de fois que `x` paraît dans `xs`. **NB** votre fonction doit être polymorphique !

Ex :

```
*Main> countFreq 'a' "Barbara"
3
*Main> countFreq 2 [1,2,3,1,2,3,1,2,4,2,3]
4
```

Solution :

```
countFreq :: Eq a => a -> [a] -> Int
countFreq x [] = 0
countFreq x (y:ys)
  | x==y = 1+countFreq x ys
  | otherwise = countFreq x ys

countFreq' :: Eq a => a -> [a] -> Int
countFreq' x xs = length [ y | y<-xs, y==x]
```