Functionnal and Logic Programming Fall 2017

S02 : Haskell (List and recursivity)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

1

f1

```haskell
-- f1 l : return the first 3 element of the list l, return the entire list if (length l < 3)
f1 :: [a] -> [a]
f1 [] = []
f1 [a] = [a]
f1 [a,b] = [a,b]
f1 [a,b,c] = [a,b,c]
f1 (a:b:c:xs) = [a,b,c]

f1' :: [a] -> [a]
f1' xs = inner xs 3 where
  inner [] n = []
  inner _ 0 = []
  inner (x:xs) n = x : inner xs (n-1)
```

fib

```haskell
-- fibonacci n : return the n-th number of fibonacci
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- using memoization, which is obviusly much more quicker
memfib :: Int -> Integer
memfib = (map fib' [0..] !!) where
  fib' 0 = 0
  fib' 1 = 1
  fib' n = memfib (n-1) + memfib (n-2)
```

2

last'

```haskell
-- last' l : return the last element of the list l
-- PRE : (null l) == false
last' :: [a] -> a
last' (x:[]) = x
last' (x:xs) = last' xs
```

delete'

```haskell
-- delete' e l : return l without the first occurence of e
delete',delete'' :: Eq a => a -> [a] -> [a]
delete' _ [] = []
delete' (e) (x:xs) = if e == x then xs else x:(delete' e xs)

-- using guards
delete'' e l
  | l == [] = []
  | e == (head l) = tail l
  | otherwise = (head l) : (delete'' e (tail l))
```

maximum'

```haskell
-- maximum' l : return the greatest element of l
-- PRE : (null l) == false
maximum' :: Ord a => [a] -> a
maximum' (x:xs) = max x xs where
  max acc [] = acc
  max acc (x:xs) = max (if x > acc then x else acc) xs
```

scalarProduct

```haskell
-- scalarProduct x y : return the scalar product of x and y
scalarProduct :: Num a => [a] -> [a] -> [a]
scalarProduct [] _ = []
scalarProduct _ [] = []
scalarProduct (x:xs) (y:ys) = (x * y) : scalarProduct xs ys
```

3

length'

```haskell
-- length' l : return the length of the list l
length' :: [a] -> Int
length' [] = 0
length' (x:xs) = 1 + (length' xs)
```

deleteAll'

```haskell
-- deleteAll' e l : return the list l without all the occurence of e
deleteAll' :: Eq a => a -> [a] -> [a]
deleteAll' _ [] = []
deleteAll' e (x:xs)
  | e == x = deleteAll' e xs
  | otherwise = x : (deleteAll' e xs)
```

toUpperString

```haskell
-- toUpperString ch : return the String (or list of char) ch in uppercase
toUpperString :: [Char] -> [Char]
toUpperString [] = []
toUpperString (x:xs) = (toUpper x):(toUpperString xs)
```

4

countVowel

```haskell
-- countVowel ch : return the number of vowel present in ch
countVowel :: [Char] -> Int
countVowel [] = 0
countVowel (x:xs)
  | elem x vowel = 1 + countVowel xs
  | otherwise = countVowel xs
  where
    vowel = "aeiouy"
```

analyseString

```haskell
-- analyseString lch : return the number of char in each string of the list lch
analyseString :: [[Char]] -> [([Char],Int)]
analyseString [] = []
analyseString (x:xs) = (x,length' x) : analyseString xs
```

analyseString2

```haskell
-- analyseString2 lch : return the number of char and the number of vowel in each string of the list lch
analyseString2 :: [[Char]] -> [([Char],Int,Int)]
analyseString2 [] = []
analyseString2 (x:xs) = (x,length' x, countVowel x) : analyseString2 xs
```