**NCCU 程式語言PL, Spring 2020**

**Assignment 2 (Basics of Haskell Programming)**

**Due date: 13:00, March 17, 2020**

**(upload your program file,** 檔名**: PL-Assignment-2.hs)**

作業寫作前可以與同學討論，但要下筆時，請自己寫，不可抄襲，

請把握學習機會。一點一滴累積下來，進步指日可待，不要浪費學分。

**1. (Simple Recursive functions, 20%)** Implement the well-known "*power*" function in two different new ways. The power function takes two arguments n and k and computes nk. Your implementation only has to work for *non-negative* integer k. The following is a straightforward implementation of this function:

power :: Int -> Int -> Int

power n k | k < 0 = error "power: negative argument"

power n 0 = 1

power n k = n \* power n (k-1)

You will implement two more ways in this part.

1. Use the standard Haskell function "product", which calculates the product (multiplication) of all elements in a list. To calculate "power n k", first construct a list with *k* elements, all being n, and then use "product". Implement this idea as a Haskell function "power1:: Int -> Int -> Int ".
2. There is a different approach to calculating the power function uses *less* computing steps: to calculate "power n k":

 If k is even, we use (n2)k/2

 If k is odd, we use n\* (nk-1)

Implement this idea as a Haskell function "power2". (Hint: Use the standard Haskell functions "even" and/or "odd")

**2. (Pattern matching, 20%)**

**(a)** Use pattern-matching with (:) and the *wildcard* pattern \_ to define a function, myButLast, that find the *last but one* element of a list. For examples;

myButLast :: [a] -> a

myButLast [1,2,3,4] = 3

myButLast ['a'[.](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:.).'z'] = 'y'

**Note:** we assume that the input list has at least two elements.

**(b)** Use pattern-matching with (:) to define a function, rev2**,** that reverses all lists of length *2*, but leaves others unchanged. Ensure that your solution works for all lists --- that is, that the patterns you use are exhaustive. For examples:

rev2 [1, 2] = [2, 1], but rev2 [1, 2, 3] = [1, 2, 3].

You may use the standard Haskell function “reverse” in the body of rev2, but you **should not** use the “length” function to determine the length of the input parameter. You may also the “@” (*as-pattern*) to simplify your code.

f s@(x:xs) = x:s is a shorthand for f (x:xs)  = x:x:xs

**3. (Tail recursion, 30%) (a)** Write a tail-recursive version of the upto function mentioned in the lecture note, call it tailUpto :: Int->Int->[Int]->[Int]. For examples:

tailUpto 3 8 [1,2] = [3,4,5,6,7,8,1,2]

tailUpto 8 3 [1] = [1]

In other words, upto m n = tailUpto m n [].

**(b)** Write a tail-recursive version of the fib function to compute the *nth* number in the Fibonacci sequence. Specifically, define the tailFib function with two accumulating parameters

tailFib :: Int->Int->Int->Int

so that fib n = tailFib n 0 1.

**4**. **(List manipulation, 30%) (a)** Write a function to determine if the elements of a list form a palindrome, palindrome [Int] -> Bool. For example:

palindrome [ 1, 2, 2, 3, 3] = False

palindrome [ 1, 2, 3, 2, 1] = True

palindrome [3] = True

palindrome [] = True

**(b)** A *permutation* of a list is another list with the *same* elements, but in a *possibly different order*. For example, [1,2,1] is a permutation of [2,1,1], but not of [1,2,2]. Write a function

isPermutation :: [Int] -> [Int] -> Bool

that returns True if its arguments are permutations of each other. For examples:

isPermutation [] [] = True

isPermutation [1,2,1] [2,1,1] = True

isPermutation [1,2,1] [2,1,2] = False

isPermutation [1,2,1] [2,1,1,2] = False

Hint: define a function, removeOnce:: Int->[Int]->[Int], that removes the *first occurrence* of an element from a list, and use it to implement isPermutation. For examples:

removeOnce 3 [1,3,5,3,4] = [1,5,3,4]

removeOnce 5 [1,2,3,3,4] = [1,2,3,3,4]

removeOnce 3 [] = []

**最後請檢查你的function names是否拼字正確。**