COMP 141: Haskell — Part 7

Instructions: In this exercise, we are going to review a bunch of Haskell structures.

(1) Define an anonymous function in Haskell that

   (a) receives a number and returns true if it is divisible to 7. Otherwise, it returns false.

   (b) receives three numbers and sums up their squares.

(2) Rewrite the following expressions with function application operator, `$`.

   (a) `filter even [1..100]`

   (b) `negate (succ (if x == 0 then 1 else 0))`

(3) Rewrite the following expressions with function composition operator, `(.)`.

   (a) `negate (succ (if x == 0 then 1 else 0))`

   (b) `filter even (map (3*) [1..10])`

(4) Define a *tail-recursive* function that receives a list of numbers and returns the summation of them. The naive version of this function (non-tail recursive version) is given in the following.

```
sum :: (Num a) => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

(5) The following function receives a list of strings and concatenates them altogether into a single string.

```
cnct :: [String] -> String
cnct [] = ""
cnct (x:xs) = x ++ (cnct xs)
```

Rewrite it to be tail-recursive. Its type would be `cnctTR :: String -> [String] -> String`, where the first input would be the accumulator.

(6) Define the tail recursive version of appendage function: `appTR`. Also, define a wrapper for it that initializes the accumulator properly.

Hint: You can use reverse function in `appTR`. Note that we have already shown that list reversal can be done tail-recursively without relying on any other function.

(7) Define tail-recursive version of `replicate` function: `replicateTR`. Also, define a wrapper for it that initializes the accumulator properly.

(8) Define tail-recursive version of `map` function: `mapTR`. Also, define a wrapper for it that initializes the accumulator properly.

(9) Define tail-recursive version of `maximum` function: `maximumTR`. Also, define a wrapper for it that initializes the accumulator properly.

(10) Define a function that creates the Fibonacci sequence of a given length. This function is supposed to be defined tail-recursively. Also, define a wrapper for it that initializes the accumulator properly.

Here is an example run of the function:

```
ghci> fibList 0
[]
ghci> fibList 1
[1]
ghci> fibList 2
[1,1]
ghci> fibList 3
[1,1,2]
ghci> fibList 10
[1,1,2,3,5,8,13,21,34,55]
```