COMP 141: Haskell — Part 4

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

(1) Run GHCi and identify the type for each of the following constructs. What does each type mean?

(a) `'&'`

(b) `"&"`

(c) `["&"]`

(d) `||` (logical or)

(e) `Data.Char.toUpper`[1]

(2) Run GHCi and identify the type for each of the following constructs. What does each type mean?

(a) `null`

(b) `reverse`

(c) `take`

(3) Define a function that has the following type: `Eq a => (a, a) -> [Char]`, meaning that

- it receives a pair and returns a string (list of characters)
- the two components of the pair must have the same type
- the type of each component of the pair is a member of type class `Eq`

(4) Define a function that has the following type: `(Ord a, Show a) => a -> a -> String`, meaning that

- it receives two inputs of any type where that type is a member of type class `Ord` and a member of type class `Show`
- it returns a string

(5) Consider the following two functions as the only available functions: `succ` and `pred`.

Write the following functions in Haskell by pattern matching on parameters, *assuming* that the arguments are positive integers.

(a) function `add :: Int -> Int -> Int` that receives two numbers and returns the summation. You are only allowed to call `succ` and `pred` in the body of `add`, along with `add` itself.

(b) function `mult :: Int -> Int -> Int` that receives two numbers and returns the product. You are only allowed to call `succ`, `pred`, and the function `add` defined above, in the body of `mult`, along with `mult` itself.

(c) function `pow :: Int -> Int -> Int` that receives two numbers `x` and `y`, and returns x^y. You are only allowed to call `succ`, `pred`, and the functions `add` and `mult` defined above, in the body of `pow`, , along with `pow` itself.

(6) Write a function, named `duplicate :: [a] -> [a]`, that receives a list of elements and duplicates the elements of that list. For example if the input is `["a","b","c"]` then the output must be `["a","a","b","b","c","c"]`.

(7) Define your own version of function `take`, called `tke :: Int -> [a] -> [a]`. Be careful about the edge cases, e.g., if the list is empty, input number is not positive, etc. Pattern match on the input list and avoid using `take`.

[1] The hierarchy of names is specifying the module in which function `toUpper` is defined. We will study it later!

(8) Define your own version of function `elem`, called `elm :: (Eq a) => a -> [a] -> Bool`. Pattern match on the input list and avoid using `elem`.

(9) Define your own version of function `Data.List.intersperse`, called `intrsprse :: a -> [a] -> [a]`. This function takes an element and a list and then puts that element in between each pair of elements in the list (if there are at least two elements in the list). For example, `intrsprse 0 [1..5]` must return `[1,0,2,0,3,0,4,0,5]`. Pattern match on the input list and avoid using `Data.List.intersperse`.

(10) Define function `flat :: [[a]] -> [a]` that receives a list of lists and flattens it into a list. For example, if the input is `[[1,2],[3,4]]` then the output must be `[1,2,3,4]`. If the input is `["hello ", "world"]` then the output is `"hello world"`. Pattern match on the input list. You cannot use function `concat`.

(11) Define function `findInd :: (Eq a) => a -> [a] -> Int` that receives an item and a list and returns the index in the list where the input item resides. If the item is not in the list, it must raise an exception. For example, `findInd 7 [5..20]` must return `2`, whereas `findInd 3 [5..20]` must raise an exception. *Note*: You cannot use the built-in function `elemIndex`. *Hint*: You may need to define a helper function that keeps track of indexes as it traverses the input list.

(12) Define function `toUppr :: Char -> Char` that receives a character and returns the uppercase if the input is lowercase. Otherwise, it returns the input character. For example, if the input is `'g'` then the output must be `'G'`. If the input is `'H'` or `'@'` then the output must be `'H'` or `'@'`, respectively. *Hint*: Check if the input is lowercase (`['a'..'z']`). If so, compute the index using the already-defined function `findInd`, and extract the associated character from `['A'..'Z']`, using that index. Do not perform pattern matching on all possible alphabetic characters! Moreover, do not user built-in `toUpper` function.