HASKELL PROGRAMMING PROBLEM SET 3

LENNART JANSSON AND BRANDON AZAD

The Types of Higher-Order Functions

Read Chapter 6 of *Learn You a Haskell*, and make sure you understand it really well. Being able to work fluently with curried functions and higher-order functions and understanding their types is really important when programming in Haskell, since they are so fundamental to the common idioms you'll use in every Haskell program.

This means it's time for another round of... Name That Type!

Problem 1. Name That Type! round 2

Give the types of all the following expressions.

(1) `(3 +) ::` _____
(2) `func1 = elem 'a'`
    `func1 ::` _____
(3) `func2 = zip [1..]`
    `func2 ::` _____
(4) `(`‘`div`‘ `10) ::` _____
(5) `map ::` _____
(6) `filter even ::` _____
(7) `show . (+ 1) ::` _____
(8) `func3 f xs = map (f . f) xs`
    `func3 ::` _____
(9) `func4 f = foldr f 0 "Hello, world!"`
    `func4 ::` _____
(10) `zipWith ($) ::` _____
(11) `func5 xs = map (λ p -> takeWhile p xs)`
    `func5 ::` _____
(12) `func6 = foldl (λ a b -> show b : a)`
    `func6 ::` _____
(13) `(.) . (.) ::` _____
(14) `(λ f g -> map (. g) . map ($ f)) ::` _____

Using Higher-Order Functions

Problem 2. Ciphering text

Add the line `import Data.Char` to the top of the file you're working in to give yourself a bunch of character processing functions. Notably, now you have access to `ord :: Char -> Int`, which gives the ASCII code of a given character, and `chr :: Int -> Char`, which does the inverse.

Write a function `vigenere :: String -> String -> String`, where `vigenere plain key` that performs the Vigenère cipher on the given plaintext with the given key. See `en.wikipedia.org/wiki/Vigenere_cipher#Description` for details if you are unfamiliar with the cipher.

```
> vigenere "ATTACKATDAWN" "LEMON"
"LXFOPVEFRNHR"
```

Problem 3. Mapping

Write a function `mapWithIndex :: (Int -> a -> b) -> [a] -> [b]` that behaves similarly to `map`, except the mapping function (the first parameter) also takes in an `Int` which is set to the index of the element being modified.

```
> mapWithIndex (λ i a -> a ++ show i) ["hello", "world"]
["hello0", "world1"]
```

Problem 4. Prelude, part 2

Let's reimplement a couple more functions from Prelude.

(1) `all :: (a -> Bool) -> [a] -> Bool`, which returns `True` if all the elements in the list satisfy the predicate, and `False` if at least one doesn't. Write this function two ways, once with manual recursion and once using preexisting higher-order functions.
(2) `foldr1 :: (a -> a -> a) -> [a] -> a`, which is like `foldr` but without a starting accumulator value. The fold starts at the right side. For example, `foldr1 (-) [5, 3, 2] == (5 - (3 - 2)) == 4`. Write this function two ways, once with manual recursion and once using preexisting higher-order functions.

Problem 5. Using folds

Implement the following functions using `foldl` or `foldr`.

(1) `product :: Num a => [a] -> a`, which computes the product of a list of numbers.
(2) `maxLength :: [[a]] -> Int`, which computes the length of the longest list out of all the lists in the input list.
(3) `compose :: [(a -> a)] -> a -> a`, which composes together of the functions in the input list.
(4) `map :: (a -> b) -> [a] -> [b]`. Yes, this can actually be done with `foldr`, but it's a little tricky!

Problem 6. Let's play a game!

This problem was inspired by Edward Yang's blog post "Let's play a game" from September 2011. (`blog.ezyang.com/2011/09/lets-play-a-game/`)

Due to the rich and expressive type system of Haskell, it's often possible to implement functions based on only their type signature, without any other information. Here's an example. Say I'm given the type signature

```
(|>) :: a -> (a -> b) -> b
```

To implement it, we have to make a function that will take an `a` and a `(a -> b)`, and give us a `b`. We realize that we can take the first parameter and feed it to the second parameter, and the result will be `(a -> b)` applied to `a` which is a value of type `b`. We're done! The implementation is:

```
a |> f = f a
```

Implement the following functions given only their type signatures.

(1) `const2 :: a -> b -> a`
(2) `fcomp :: (a -> b) -> (b -> c) -> a -> c`
(3) `appc :: ((b -> a) -> c) -> a -> c`
(4) `on2 :: (c -> d -> e) -> (a -> c) -> (b -> d) -> a -> b -> e`
(5) `fmapC :: (a -> b) -> ((a -> r) -> r) -> (b -> r) -> r`

(6) `bindC :: ((a -> r) -> r) -> (a -> (b -> r) -> r) -> (b -> r) -> r`

Challenge Problems

Problem 7. Fibonacci

Implement `fib :: Integer -> Integer`, where `fib n` gives the nth Fibonacci number. Make sure your program runs in linear time. That is, you should be able to calculate `fib 200` almost instantaneously.

```
> fib 0
0
> fib 1
1
> fib 200
280571172992510140037611932413038677189525
```

Problem 8. Magic

Implement `foldl` in terms of `foldr`.