Haskell – Assignment 3

This assignment is worth 50 points.

Notes:

- You *must* write type signatures for all functions/values you define. If you're ever confused about what the best type signature for a function might be, try defining the function body without the type signature, then load your file into `ghci` and run `:t <function>` to see what type `GHC` infers for your function `<function>`.
- This assignment will be graded with unit tests, so for each problem **be sure you use the exact function name given in the problem statement to define your function**.
- **No credit** will be given if you submit an assignment that will not load the test file. So, if you cannot figure out how to define a function, **set the right-hand side of your function to `undefined`**. Example: `myFunc = undefined`.
- Please download this source file and add your answers to it. Do *not* change the name of the file. This is the file you should submit when you're finished. This is also linked in the schedule on the course website if the link here doesn't work.
- **Problems 1-3 are due Tuesday, November 7 at midnight on Canvas.**
- **Problem 4 is due Thursday, November 9 in class.**
- If you're confused about how to fit certain functions together, look at the type signatures! That's one of the most useful tools you have when using Haskell.

Problem 1 – `map` and `filter` (12 pts.)

4 pts. each

The source file included with this assignment contains the following data declaration for a datatype called `Artist`:

```
data Artist = Artist Name Genres
  deriving Show
type Name   = String
type Genres = [Genre]
type Genre  = String
```

Recall that the `type` keyword declares a *type synonym*.

The `Artist` type represents a solo musician or band and has a single constructor which takes two arguments: the name of the artist and a list of genres associated with that artist.

In the example output for the problems below, assume the following value bindings have been declared (these are also in the provided source file):

```
band :: Artist
band = Artist "Talking Heads" ["New Wave", "Post Punk", "Pop Rock", "Funk Rock"]

artists :: [Artist]
artists = [
    Artist "Talking Heads" ["New Wave", "Post Punk", "Pop Rock", "Funk Rock"],
    Artist "Devin Townsend" ["Progressive Metal", "Ambient", "New Wave"],
    Artist "Brand New" ["Pop Punk", "Emo", "Alternative Rock", "Indie Rock"],
    Artist "Brian Eno" ["Art Rock", "Ambient", "Electronic"],
    Artist "The Front Bottoms" ["Indie Rock", "Folk Punk", "Anti-Folk"],
    Artist "Grimes" ["Electropop", "Dream Pop", "Synthpop"],
    Artist "My Bloody Valentine" ["Shoegaze", "Noise Pop", "Post Punk"],
    Artist "David Bowie" ["Art Rock", "Pop Rock", "Glam Rock", "New Wave"]
  ]
```

a. `getName`, `getGenres`

Write two functions, `getName` and `getGenres`, that each accept an `Artist` and return the artist's name and list of genres, respectively.

```
> getName band
"Talking Heads"

> getGenres band
["New Wave", "Post Punk", "Pop Rock", "Funk Rock"]
```

b. `getSortedNames`

Write a function called `getAllArtistNames` that accepts a list of `Artist`s and returns a *sorted* list of all of the artists' names. *You must use `map` in your answer.* This is also a good opportunity to practice function composition with `.`, but you're not required to.

Note: You'll want to import the `sort` file from `Data.List`. We haven't discussed importing in lecture, but Chapter 7 of the book gives an overview of it.

```
getSortedNames artists
> ["Brand New","Brian Eno","David Bowie","Devin Townsend",
   "Grimes","My Bloody Valentine","Talking Heads",
   "The Front Bottoms"]
```

c. `filterByGenre`

Write a function called `filterByGenre` that accepts a list of `Artist`s and a single genre string and returns all of the `Artist`s with that genre. *You must use the `filter` function in your answer.*

```
> filterByGenre artists "New Wave"
[Artist "Talking Heads" ["New Wave","Post Punk","Pop Rock","Funk Rock"],
 Artist "Devin Townsend" ["Progressive Metal","Ambient","New Wave"],
 Artist "David Bowie" ["Art Rock","Pop Rock","Glam Rock","New Wave"]]

> filterByGenre artists "Deathcore"
[]
```

Problem 2 – $ and . (6 pts.)

a. Function Application with $ (4 pts.)

Write a function called `multTableRow` that accepts an `Int` and returns one row of a multiplication table. *You must use the* `$` *operator in your answer* (check out Chapter 6 of the textbook, it has a good example of that).

```
> multTableRow 3
[3,6,9,12,15,18,21,24,27,30]
> multTableRow 5
[5,10,15,20,25,30,35,40,45,50]
```

I recommend you do this in two steps:

First, use the `map` function to create a list of *partially applied functions*. If you could print out this list (which you can't), it'd look something like: `[(1*), (2*), ..., (10*)]`.

Once you have that list, use `map` again to call each of those functions on the input value. This is where you'll use `$`. Remember to look carefully at the types if you get confused (e.g. you might find it useful to look at the type of the list you created in the previous paragraph, along with the type of `map` and `$`).

b. Function Composition with . (2 pts.)

Write a function called `doubleNegate` that takes in a list of numbers and returns the result of doubling and negating every element of the list. *You must use* `.` *in your answer.*

Hint: You may find the `negate` *function useful.*

```
> doubleNegate [1,2,3]
[-2,-4,-6]
> doubleNegate [0, 1.5, 5, 10]
[-0.0,-3.0,-10.0,-20.0]
```

Problem 3 – `fold` Applications (12 pts.)

Note: If you have trouble in this section, be sure to refer to the section of Chapter 6 of the textbook that discusses folds.

You must use one of the folding functions (`foldl`, `foldr`, `foldl'`, `...`) in your solutions to the problems in this section. Note that some of the fold functions need to be imported from `Data.List`, while others are available by default via `Prelude`

a. `totalDiscount` (4 pts.)

Write a function named `totalDiscount` that takes a discount percent (as a decimal) and a list of prices, and returns the total amount saved by the discount.

```
> totalDiscount 0.1 [1]
0.1
> totalDiscount 0.1 [1, 4]
0.5
> totalDiscount 0.5 [1, 2, 3]
3.0
```

b. `discountedItems` (8 pts.)

Write a function named `discountedItems` that takes a discount percent and a list of prices, and returns the total price for each item after the discounted has been applied.

Hint: Look carefully at the type signature for whichever `fold` function you decide to use and try to match the type variables to the types we have in this problem.

```
> discountedItems 0.1 [4, 5, 6]
[3.6, 4.5, 5.4]
```

Problem 4 – `fold` Reductions (20 pts.)

You should print this problem, write your answer on the pages, and turn it in on Thursday, November 9.

Let's explore the difference between `foldl` and `foldr`.

The top answer to this StackOverflow question is probably my most revisited SO answer about something in Haskell. The question itself isn't as important as the answer – read through the bullet points at the top and the first paragraph after those points.

Here are the definitions of `foldl` and `foldr`:

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Consider the following expression:

```
foldl (&&) True [False | _ <- [1..]]
```

Try running this in `ghci`.

Question a (1 pt.): What happens?

Answer:

Now replace `foldl` with `foldr` and try running it again.

```
foldr (&&) True [False | _ <- [1..]]
```

Question b (1 pt.): What happens this time?

Answer:

Now comes the hard question: *Why does this happen?* To figure that out, let's try to reduce these expressions by hand. In the two sections below, use the given rules to write out the reduction steps. Write out enough steps to make it clear what's happening – *you won't be able to get to the end*, of course, since we're dealing with infinite recursion.

For convenience, we'll write the infinite list `[False | _ <- [1..]]` as `[False..]`, even though the latter is not valid Haskell.

Reduce the following expression: `foldl (&&) True [False..]` for a few steps. *You must apply rule 2 at least two times before you stop.* Note that the only requirement is that you apply rule 2 at least twice – you may not end up using

rules 3 or 4 at all, it's kind of up to you, but using those rules may help you answer the final question for this problem.

Initial expression: `foldl (&&) True [False..]`

Rules

1. `[False..] => False:[False..]`
2. `foldl f z (x:xs) => foldl f (f z x) xs`
3. `True && False => False`
4. `False && False => False`

Steps (6 pts.)

Now do the reduction steps for the expression: `foldr (&&) True [False..]`. This time, you should get to a final expression (the same value that `ghci` gave you), as this one stops after a few steps. You should end up using all of the provided rules this time.

Initial expression: `foldr (&&) True [False..]`

Rules

1. `[False..] => False:[False..]`
2. `foldr f z (x:xs) => f x (foldr f z xs)`
3. `False && _ => False`

Steps (6 pts.)

Remember that, under the hood, Haskell is doing these types of reductions for us. That's the whole reason we're doing them – they help understand how the execution is taking place, and give you a bit of insight when writing your own code. So, finally: **explain why the first case, with `foldl`, runs forever and the second, with `foldr`, does not. (6 pts.)**