Haskell: Assignment 1

Notes:

- For all of the problems below, you *must* write a type signature for any function/value you write in your code.
- You should turn in the **first two problems** on Canvas in a `.hs` file that has the functions you define for each part. These are due by the night of **Tuesday, October 24** (midnight).
- You should *print out* ***Problem 3***, write your answer on that sheet, and turn it in on **Thursday, October 26** in class.
- You can test your code yourself by loading your `.hs` file into `ghci`.

Problem 1 – Factorial

For each of the following sub-problems, you are going to write a `factorial` function that meets a specific requirement. The following is relevant to both parts:

- The input to the function is an `Int`, n
- The output of the function is the mathematical factorial of the input, $n!$, also an `Int`
- You *do not* have to handle negative input values – assume the input is 0 or greater.

a. Guard Expression

Write the `factorial` function using a *guard expression.* You may want to refer to the fibonacci function `fib` in the lecture notes.

b. Pattern Matching

Write the `factorial'` function using *pattern matching* (and no guards). You may want to refer to the second fibonacci function `fib'` from the lecture notes.

Note the `'` at the end of the function name that differentiates it from the name of the function in part ***a****.*

Problem 2 – `Maybe`

Haskell does not have an untyped empty value like `NULL` in C++, `nil` in Ruby, `None` in Python, `null` in Java, etc. Instead, Haskell uses data types and polymorphism to reach a similar end. The type used for this is `Maybe`, which is defined as:

```
data Maybe a = Nothing | Just a
```

`Nothing` is Haskell's equivalent of `NULL`. But this type definition looks kind of funny, primarily because of the `a`.

Recall Haskell's general list type, `[a]`. The `a` is a *type variable*, so one way to read `[a]` is "a list of elements of any type". The `a` in `Maybe a` is also a type variable. So, just as we can have a list of `Int`s with the type `[Int]`, we can also have a `Maybe Int`.

Now turning to `Maybe`'s constructors, we have `Nothing` and `Just a`.

- `Nothing` is simply a constructor with no arguments – by default, its type binding is `Nothing :: Maybe a`, but within a particular context it may be `Nothing :: Maybe Int`, `Nothing :: Maybe String`, etc.
- The `Just a` constructor tells us that we can use the `Just` constructor with an *argument of any type* (hence the type variable `a`).

Let's look at a few examples using `Maybe`'s constructors.

If we bind `Nothing` to a symbol `n` and we don't assign a type, like so:

```
n = Nothing
```

Then the compiler will infer `n`'s type to be `Maybe a`, since it has no way to narrow down the type variable `a` any further.

We could also force `n` to be of a more specific type:

```
n :: Maybe Int
n = Nothing
```

or

```
n = Nothing :: Maybe Int
```

The compiler can infer a bit more about a binding that uses `Just`:

```
j = Just "hi"
```

In this case, Haskell will infer that `j`'s type binding must be `j :: Maybe String`, since we provided a `String` argument to the `Just` data constructor.

To see where `Maybe` might be useful, consider the function `head :: [a] -> a`, which returns the first element of a list. What if the input list is empty? `head` wouldn't be able to return a value of the expected type. In fact, if you called `head []` in `ghci`, you'd get an error.

Now imagine a function `headMaybe` with the type binding:

```haskell
headMaybe :: [a] -> Maybe a
```

Example function calls:

```
headMaybe [1, 2, 3]
=> Just 1
headMaybe ['a', 'b', 'c']
=> Just 'a'
headMaybe []
=> Nothing
```

From these examples, we can see that `headMaybe` returns Just <first_element> on success, and `Nothing` on failure.

Define the function `headMaybe` so that it behaves as described above.

Hint: All you should use on the right-hand side (RHS) of your definition are the constructors for `Maybe` as well as the `head` function mentioned above.

Problem 3 – Reduction

Recall the `Peano` number data type example from lecture:

```
data Peano = Zero | Succ Peano
    deriving Show
```

In lecture, we wrote a particular definition for a function `add` that added two `Peano`s together. Our definition was longer than it needed to be, though. Here's a partial definition for a different form of the `add` function:

```
add :: Peano -> Peano -> Peano
add Zero p = p
```

a. Add *one* more case to the `add` function's definition that completes the definition and will successful add all `Peano` numbers. Write the line below. *Hint: Think about the two fundamental cases in a recursive function.*

b. Using your completed definition of `add`, write out the reduction steps for the expression `add two one`, where `two = Succ (Succ Zero)` and `one = Succ Zero`. Be sure to define your *reduction rules* before you do the reduction steps (there should be **4 rules** in this case).

Rules:

Steps: