Functional Programming I (CS4620) Assignment 1

Getting Started (Due: October 8. Marks: 5)

Introduction

This assignment is about getting used to `ghc` and `ghci` and about learning different techniques to implement functions. When you're finished with this assignment you should know: (1) how to define functions on top of existing functions, (2) how to define them using pattern-matching, (3) how to define them using lambda expressions, and (4) how to define them using partial applications.

Please remember the following.

- Every function definition should include a proper type declaration. Not only is adding them a proper form of documentation but it is also a good exercise.

  Please note that you should infer the type of the function yourself, *based on what we have studied in class.* So, e.g. if `ghci` tells you the type of one of your functions is `Foldable t => t [a] -> [a]` because it is equivalent to `concat` then you shouldn't use that type but use `[[a]] -> [a]` instead because we never studied the class `Foldable`.
- Every function should include a brief explanation about the implementation technique.
- Unless specified otherwise, you should only use techniques that we've studied in the lectures leading up to this assignment. These techniques correspond to Chapters 1–4 from the book. In addition you are also allowed to use function composition.

Assignment Details

A faraway forest has two kinds of inhabitants: *knights* and *knaves*. Knights always tell the truth and knaves always lie [Smullyan 1990].

For the purpose of this assignment we shall model the inhabitants as functions that take a `Bool` argument (the truth value of the question) and return a `Bool` value (the answer). So, for example if you give a value to a knight, the knight will return the same value to you. On the other hand, a knave will return the complement of the value.

For this assignment you will provide *several* implementations of knights, knaves, and functions that take knight or knave arguments. To make things challenging, you are not allowed to use the conditional statement.

The following are the first tasks.

- Define a function called `knight1` for a knight that just returns its argument in the right-hand side of the function's definition:

```
knight1 q = <your solution here>
```

- Define a function called `knave1` for a knave that just returns the complement of its argument in the right-hand side of the function's definition. You should compute the complement using a built-in Haskell function.

```
knave1 q = <your solution here>
```

- Define a function called `knight2` for a knight that uses pattern matching.
- Define a function called `knave2` that uses a lambda-expression.
- Define a function called `knave3` for a knave that uses a partial application.
- Define a function called `inhabitants` that returns a list consisting of the functions `knight1`, `knight2`, `knave1`, `knave2`, and `knave3`.
- Define a function called `getInhabitant` that takes in an `Int`, *i*, and returns the *i*th member from the list `inhabitants`. (There is no need to add error handling.)
- Implement a function called `ask` that takes in an `Int`, *i*, and a `Bool`, *b*, and returns the value that is returned if you call the *i*th member of the list `inhabitants` and pass it the argument *b*. (There is no need to add error handling.)

Before you continue, please make sure all your functions have a type definition and a short explanation about the implementation technique.

Remember that in mathematics $f \circ g$ is a function that takes a single argument. The function first applies g and then applies f to the result of this application:

$$(f \circ g)(x) = f(g(x)) .$$

In Haskell we denote function composition with the dot operator:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
-- equivalently
(.) :: (b -> c) -> (a -> b) -> a -> c
```

For example `(succ . succ) 0 :: Int` results in 2.

- Using function composition, implement a function called `double_negation` that takes a `Bool`, *b*, and returns `knave1( knave1(` *b* `) )`.
- Is the function `double_negation` equivalent to `knave1`? If yes, why? If not, please describe the function `double_negation`.

Before you continue, please make sure all your functions have a type definition and a short explanation about the implementation technique.

The purpose of the rest of this assignment is to implement "interrogate" functions that pose cunning questions that makes the inhabitants from the forest provide the correct answer to any question. E.g. `interrogate knight1 True` should return in `True`, `interrogate knave1 False` should return in `False`, and so on. The "interrogate" functions should do this by returning their result using a top-level call to their inhabitant argument.[1] You are not allowed to implement the "interrogate" functions with decision making or pattern-matching techniques.

[1]So defining `interrogate _ b = b` is not allowed.

In your “interrogate” functions, the inhabitant should be the first argument and the `Bool` should be the second argument. The following are the requirements for the functions:

- `interrogate1` should use both arguments in the right-hand side of the definition;
- `interrogate2` should use a lambda-expression;
- `interrogate3` should use a partial application.

Before you submit this assignment, please make sure all your functions have a type definition and a short explanation about the implementation technique.

Submission Details

- Your program should start with a comment like the following:

```
{-
 - Name: Fill in your name.
 - Number: Fill in your student ID.
 - Assignment: Assignment Number.
 -}
```

- Use the CS4620 `moodle` site to upload your program as a single *.tgz* archive called *Lab-1.tgz* before 23.55pm, October 8, 2017. To create the *.tgz* archive, do the following:
  - ⋆ Create a directory `Lab-1` in your working directory.
  - ⋆ Copy `Main.hs` into the directory. Do not copy any other files into the directory.
  - ⋆ Run the command ‘`tar cvfz Lab-1.tgz Lab-1`’ from your working directory. The option ‘v’ makes `tar` very chatty: it should tell you exactly what is going into the `.tgz` archive. Make sure you check the `tar` output before submitting your archive.
  - ⋆ Notice that file names in `Unix` are case sensitive and should not contain spaces.
- Notice that the format is `.tgz`: do *not* submit `zip` files, do *not* submit `tar` files, do *not* submit `bzip` files, and do *not* submit `rar` files. If you do, it may not be possible to unzip your assignment.
- Marks are deducted for poor choice of variable names and/or poor layout.
- As explained in lecture 4, you should make sure your assignment submission should have a `Main` class with a `main` in it. The `main` should be the main thread of execution of the program.
- No marks shall be awarded for scripts that do not compile.

References

Smullyan, R. [1990]. *To Mock a Mocking Bird.* Oxford University Press. ISBN: 0-19-286095-x.