Functional Programming I (CS4620) Assignment 5

Class Design (Due: 24 November. Marks: 10)

1 Introduction

This is a double assignment, which is worth twice the regular amount of marks. The assignment is about implementing a class in `Haskell`, instantiating the class with an existing type, and extending existing classes. The main purpose of the class is to provide functionality to check hypotheses about values from types that inhabit the class.

Please remember that your programs should be properly commented. Also please note that all function definitions in your programs should include a proper type declaration. Not only is adding them a proper form of documentation but it is also a good exercise.

2 Assignments Details

For this assignment you will implement a class `Testable` for checking hypotheses about values belonging to types that are `Testable`. The assignment has five parts. Using comments similar to the following, you should indicate where the implementation of these parts start.

-- *PART* <*NUMBER*>

2.1 First Part

For the first part of this assignment you should define a user-defined data type called `TwoValued` consisting of two values: `One` and `Two`. You should make sure that values of the data type are

comparable with `(==)` and with `(/=)`;

]ordered] with `(<)`, with `(<=)`, ...; and

convertible to string (showable).

As suggested by the names, the value `One` should be the smaller value and the value `Two` should be the larger value.

The `Bounded` class defines functions `minBound` & `maxBound`, which define the lower & upper limit for types belonging to the class. Provide statements that make `TwoValued` an instance of the class `Bounded`. Doing this may require an `instance` declaration that overrides the two functions for `TwoValued` values.

2.2 Second Part

For the second part you should define a class `Incrementable` that provides the following function:

`next a` returns the smallest value such that `a < next a`. E.g. `next False` is `True` but `next True` and `nxt Two` are not defined.

Inhabit the class with the `TwoValued` data type. To do this, you have to write an `instance` declaration that overrides the function `next` for `TwoValued` values.

2.3 Third Part

For the third part you will implement a class `Testable` for testing predicates about values from types that are `Incrementable` and orderable (`Ord`). The `Testable` class defines the following function.

`check p start end` returns `True` if and only if `p a` holds for all values `a` such that `start <= a` and `a <= end`.

E.g. if `ot = [One,Two]` then we have the following for the `TwoValued` data type:

- `[check (==One) first last | first <- ot, last <- ot]` is `[True,False,True,False]`; and
- `[check (==Two) first last | first <- ot, last <- ot]` is `[False,False,True,True]`.

Please make sure you define a default implementation for the function `check`. This class took me six lines.

Inhabit the class with the `TwoValued` data type. To do this, you have to write an `instance` declaration. There is no need to override the function `check` because it has a default implementation.

2.4 Fourth Part

For the fourth part you will make lists of type `[a]` instances of the `Incrementable` class for values of type `a` that are in the `Incrementable` class, in the `Ord` class, in the `Enum` class, and in the `Bounded` class. Hint: the implementation should start as follows.

```
instance (Incrementable a,Eq a,Ord a,Bounded a) => Incrementable [a] where
```

Please remember the part between the keyword `instance` and the implication arrow (=>) is called the *context* for the type variabls on the right hand side of the implication arrow.

Implementing the class took me five lines, which included a line for error-handling, which wasn't really needed. For your implementation, please follow the suggestions in the remaining paragraphs.

Define `next = reverse . next' . reverse`. Notice that because of the context of the class, the function `next'` operates on lists with values from types that are in the `Incrementable` class, in the `Ord` class, in the `Enum` class, and in the `Bounded` class. Because of this you can increment type–`a` values that are less than `maxBound` with `next`, you can compute the smallest possible type–`a` value with `minBound`, and you can compare type–`a` values with ==, with <, and so on. The result of the function `next'` is the "increment" of a reversed list. E.g. for a type that has values $v_1 < \cdots < v_k$, we can define a "increment" function on reversed singleton lists:

- next’ [v_1] = [v_2],
  next’ [v_2] = [v_3],
  ...
  next’ [v_{k-1}] = [v_k].

We can do the same for reversed lists with two values:

- next’ [v_1, v_1] = [v_2, v_1],
  next’ [v_2, v_1] = [v_3, v_1],
  ...,
  next’ [v_k, v_1] = [v_1, v_2],
  next’ [v_1, v_2] = [v_2, v_2],
  next’ [v_2, v_2] = [v_3, v_2],
  ...,
  next’ [v_k, v_2] = [v_1, v_3],
  ...
  next’ [v_{k-2}, v_k] = [v_{k-1}, v_k], and
  next’ [v_{k-1}, v_k] = [v_k, v_k].

The function `next’` can be implemented in four lines (including the type).

Please note that the previous lines suggests a length-ℓ list represents a sequence of ℓ digits, where the digits have k possible values. There are exactly k^ℓ such sequences, which should be useful for testing. Also please note that the previous lines the significant digits are at the start of the (non-reversed) list. You should make sure you also do this.

2.5 Fourth Part

The fifth part of this assignment implements the `main`, which should be as follows.

```
main = do putStrLn $ show $ test ([One,Two,Two] ==)
          putStrLn $ show $ test ([One,Two,Two] /=)
          putStrLn $ show $ test ([One,Two,Two]  <)
          putStrLn $ show $ test ([Two,Two,Two] >=)
          putStrLn $ show $ test (>= [Two,Two,Two])
     where test p = [int p bs [Two,Two,Two] | bs <- bss ]
           int p a a' = if check p a a' then 1 else 0
           bs = [One,Two]
           bss = [[a,b,c] | a <- bs, b <- bs, c <- bs ]
```

Please copy-and paste this code into your source file. Do *not* (as in “do *not*”) provide a different implementation of the `main` function.

2.6 Final Comments

All user-defined data types, classes, and function should be implemented from scratch, without libraries. Please do *not* use list indexing (`!!`)[1] as all list operations can be implement using recursion

[1]This means you also shouldn’t redefine list indexing.

and simple pattern-matching. Your implementation should be standard `Haskell`, so you should not use special `ghc` extensions.

3 Submission Details

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

  ```
  {-
   - Name: Fill in your name.
   - Number: Fill in your student ID.
   - Assignment: 05.
   -}
  ```

- Use the CS4620 `moodle` site to upload your program as a single *.tgz* archive called *Lab-5.tgz* before 23.55pm, 24 November, 2017. To create the *.tgz* archive, do the following:
  - ⋆ Create a directory `Lab-5` in your working directory.
  - ⋆ Copy `Main.hs` (or `Main.lhs`) into the directory. If your implementation requires other user-defined `Haskell` scripts, you should also copy them into the directory. Do not copy any other files into the directory.
  - ⋆ Run the command '`tar cvfz Lab-5.tgz Lab-5`' 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` command's output before submitting your archive; alternatively, use `tar -t` or `tar --list`.
  - ⋆ Note that file names in `Unix` are case sensitive and should not contain spaces.
- Note 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 may be deducted for poor choice of identifier 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.