Functional Programming I (cs4620) Assignment 4

Implementing Lists (Due: 15 November. Marks: 5)

1 Introduction

This assignment is about implementing polymorphic, user-defined data types.

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 representation for *polymorphic* lists. You will learn how to define user-defined recursive algebraic data types and how to implement functions on these new types by pattern matching on the constructors of the data types.

The following are the requirements.

- Define the an algebraic data type called `List` for our lists. The `List` data type should not be built on top of the `[]` type.
- Implement a polymorphic function called `insert` for inserting an element into an existing `List`. The `insert` function takes two arguments: the first is the `List` and the second is the element. The `insert` function should insert its value argument in the "ordered" position of its list argument: the smaller elements should be at the start of the list the larger elements should be at the end of the list. *Hint: the values in the list are in the `Ord` class; you can enforce this by giving the function a proper context.*
- Note that if we only insert items in a `List` with the function `insert`, then our user-defined lists will be ordered from least significant to most significant. If `insert` is the only function that inserts elements into `List`-based lists, then you may assume that every `List`-based list is ordered.
- Implement functions `foldr'` and `foldl'` which are to `List` what `foldr` and `foldl` are to `[]`.
- Implement a function called `pretty_print` for pretty printing a `list`, i.e. for converting a list to string. The function `pretty_print` should use square brackets at the start and end of the list and should use commas to delimit the elements. E.g. an empty list will be "pretty printed" as `"[]"` and a list consisting of a 1, then a 3, and then a 4 will be pretty printed as `"[1,3,4]"`. Please note that there are no superfluous commas in the pretty printed lists.
- Using `foldr`, implement a function called `create_from_list` that creates a `List` from a standard list.

All user-defined data types and functions should be implemented from scratch, without libraries. Your implementation should be standard `Haskell`, so you should not use special `ghc` extensions.

Your `main` should be as follows:

```
> main = putStrLn $ pretty_print $ create_from_list list
>      where list = [1,2,3,5,4] :: [Int]
```

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: 04.
 -}
```

- Use the CS4620 `moodle` site to upload your program as a single *.tgz* archive called *Lab-4.tgz* before 23.55pm, 15 November, 2017. To create the *.tgz* archive, do the following:
  - Create a directory `Lab-4` 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-4.tgz Lab-4`' 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.