Exercise Sheet Four : Higher-Order Functions and Declared Data Types

The aim of this tutorial is for you to gain experience using higher-order functions as an abstraction mechanism to encourage code reuse via map, filter and fold. It also aims for you to understand the use of algebraic data types in Haskell.

Attempt the exercsise in your own time before Tuesday's lab and use that session for obtaining feedback and for asking questions related to the exercises.

Exercise One

For each of the following functions give definitions for them using map, filter and/or foldr.

- Decide if all elements of a list satisfy a predicate
  `all :: (a -> Bool) -> [a] -> Bool`
- Decide if any element of a list satisfies a predicate
  `any :: (a -> Bool) -> [a] -> Bool`
- Select the initial elements from a list while they satisfy a predicate
  `takeWhile :: (a -> Bool) -> [a] -> [a]`
- Remove the initial elements from a list while they satisfy a predicate
  `dropWhile :: (a -> Bool) -> [a] -> [a]`

Exercise Two

Using `foldl` define a function `dec2Int :: [Int] -> Int` that takes a list of digits as a representation of a decimal number and converts it in to an integer representing that number. For example `dec2Int [1,2,3,4] = 1234`.

Exercise Three

Define the higher-order function `curry` that takes a function that converts functions on pairs in to their curried form that accept their arguments one by one. Conversely, define the function `uncurry` that provides the inverse to this. Write down the types of these functions first to guide you.

Exercise Four

Consider the recursive pattern described by the higher-order function below:

```
unfold p h t x | p x       = []
               | otherwise = h x : unfold p h t (t x)
```

This is essentially a list generating pattern. The predicate `p` determines the termination condition, the argument `h>` describes how to insert the next element in to the list and `t` describes how the remainder should be processed.

Use `unfold` to define the following functions:

- `int2bin` - that converts an integer to a binary representation as a list of binary digits.
- `chop` - that takes a string and chops it in to a list of strings of a given length.
- `map` - the standard map function.
- `iterate` - that takes a function `f` and a value `x` and produces the infinite list
  `[ x, f x , f ( f x) , f ( f ( f x)) , ... ]`

Exercise Five

Define a function `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` that accepts two functions and alternately applies them to successive elements in a list. For example,

```
> altMap (+10) (+100) [0,1,2,3,4]
[10,101,12,103,14]
```

Exercise Six

Remember the Luhn algorithm from Sheet 2 Exercise 10 for validating 4-digit bank card numbers? Now implement this algorithm for bank card numbers of any length. See if you can define the function `luhn` as a composition `(.)` of functions.

Exercise Seven

A binary search tree is a tree in which every node contains only smaller values in its left subtree and larger values in its right subtree. A tree is balanced if the length of every path from the root to each leaf differs by at most one.

For the given definition of type Tree

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show
```

define a function `toTree :: Ord => [a] -> Tree a` that constructs a balanced, search tree from a given list.

Exercise Eight

Consider the recursive data type

```
data Nat = Zero | Succ Nat deriving (Eq,Ord,Show,Read)
```

This allows us to represent any natural number as sequences of the successor operation

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
```

Define functions, `even :: Nat -> Bool` and `odd :: Nat -> Bool` that determine whether the given Nat is even or odd. Now define functions `add,mult :: Nat -> Nat -> Nat` that implement addition and multiplication respectively.

Exercise Nine

Consider a variation on the `Nat` datatype of recursively defined integers. We have a successor operation (add one) and a predecessor operation (subtract one).

```
data RInt = Zero | Succ RInt | Pred RInt deriving Show
```

We say that a value of type `RInt` is in *normal* form if it does not contain a mix of both `Succ` and `Pred` constructors. Write a function `normalise :: RInt -> RInt` that converts any `RInt` value in to a normal form of the same value.

Provide definitions of `odd,even,add,mult` as above for this datatype. You may find it useful to use `normalise` to do this.