Exercise Sheet Six : Equations and Evaluation

The aim of this tutorial is to gain skills in performing equational reasoning for Haskell programs and to understand the concepts of reduction, redexes and lazy evaluation.

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

Define the append function as

```
   [ ] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

and use equational reasoning to show that

- `xs ++ [ ] = xs` and
- `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Exercise Two

Remember the data type `data Nat = Zero | Succ Nat`. We can use this to act as a simple counter for functions. For example, we can write

```
replicate :: Nat -> a -> [a]
replicate Zero _ = []
replicate (Succ n) x = x : replicate n x
```

We wish to prove that `replicate` produces a list whose elements are all equal to the element given to replicate. To do this we can define the function

```
all :: (a -> Bool) -> [a] -> Bool
all p [] = True
all p (x:xs) = p x && all p xs
```

and prove that `all (== x) (replicate n x) = True` for any x and any n. Write this proof. What assumptions do you need to make about (==) in order for the proof to hold? Are these valid assumptions?

Exercise Three

Prove that for any list we have `take n xs ++ drop n xs = xs` where

```
take Zero _  = []
take _ [] = []
take (Succ n) (x:xs) = x : take n xs

drop Zero xs = xs
drop _ [] = []
drop (Succ n) (_:xs) = drop n  xs
```

Exercise Four

Define the following functions over `Nat`

- `even :: Nat -> Bool` that determines whether a `Nat` represents an even number.
- `double :: Nat -> Nat` that doubles a given `Nat`

Prove that `even (double n) = True` for any `n`.

Exercise Five

Using the data type `data Tree a = Leaf a | Node (Tree a) (Tree a)` show that in any such tree the number of leaves is always one greater than the number of nodes.

Exercise Six

Verify the functor laws for the functors `Maybe` and `Tree a` where

```
instance Functor Maybe where
 -- fmap :: (a -> b) -> Maybe a -> Maybe b
 fmap g Nothing = Nothing
 fmap g (Just x) = Just (g x)

instance Functor Tree where
 -- fmap :: (a -> b) -> Tree a -> Tree b
 fmap g (Leaf x) = Leaf (g x)
 fmap g (Node l r) = Node (fmap g l) (fmap g r)
```

Exercise Seven

Identify the redexes in the following expressions and categorise them as innermost, outermost, neither or both

- 1 + (2 * 3)
- (1 + 2) * (2 + 3)
- fst (1 + 2 , 2 + 3)
- (\x -> 1 + x) (2 * 3)

Exercise Eight

Define `mult = \x -> (\y -> x * y)` and show all of the evaluation steps that Haskell would take when evaluating `mult (mult 3 4) 5`

Exercise Nine

Use a list comprehension to define the infinite Fibonacci sequence 0,1,1,2,3,5,8,13,21,34 ... You will find it useful to use `zip` and `tail` and you may want to use the type Integer rather than Int.

Exercise Ten

Recall the functions `repeat :: a -> [a]` and `take :: Int -> [a] -> [a]` for lists? They produce an infinite list of copies of the given element and a prefix of a given size of a list respectively. We can compose them to define `replicate n a` that produces a fixed size list of copies of `a`.

Write analagous functions

- `repeatTree :: a -> Tree a`
- `takeTree :: Int -> Tree a -> Tree a` and

- `replicateTree :: Int -> a -> Tree a`

that work on the `Tree` data type. Make sure you define the latter function modularly.