Quiz 2

CS 613 – Design and Implementation of Functional Programming Languages

Date 6th October 2016 Total Marks 45

1. Consider the datatype:

```
data Tree a b = Node a (Tree a b) (Tree a b) | Leaf b
```

Make Tree an instance of Functor class. What is the value of the expression `fmap (\x->x*x) (Node "a" (Node "b" (Leaf 4) (Leaf 6)) (Leaf 7))`?. (5 Marks)

2. Write a function `main` which will read text from a file called `input` and insert a blank line between successive lines of the text. Compile your program using ghc. If the executable is q2, invoke the executable as `q2 < input`. (7 Marks)

3. Consider the following datatype:

```
data CMaybe a = CNothing | CJust Int a deriving Show
```

and the following instance declaration:

```
instance Functor CMaybe where
  fmap f CNothing = CNothing
  fmap f (CJust counter x) = CJust (counter+1) (f x)
```

Show that this declaration does not satisfy the Functor laws. (5 Marks)

4. Declare Tree to be an instance of Applicative so that we can write an expression like:

```
Node 1 (Leaf (*1)) (Node 2 (Leaf (*2)) (Leaf (*3))) <*>
Node 10 (Leaf 4) (Node 20 (Leaf 5) (Leaf 6))
```

The answer of this should be:

```
Node 1 (Node 10 (Leaf 4) (Node 20 (Leaf 5) (Leaf 6)))
     (Node 2 (Node 10 (Leaf 8) (Node 20 (Leaf 10) (Leaf 12)))
             (Node 10 (Leaf 12) (Node 20 (Leaf 15) (Leaf 18))))
```

(8 Marks)

5. Now declare Tree to be an instance of Monad, so that:

```
do
  x <- Node 10 (Leaf 4) (Node 20 (Leaf 5) (Leaf 6))
  Node 1 (Leaf (x*1)) (Node 2 (Leaf (x*2)) (Leaf (x*3)))
```

evaluates, once again to:

```
Node 1 (Node 10 (Leaf 4) (Node 20 (Leaf 5) (Leaf 6)))
     (Node 2 (Node 10 (Leaf 8) (Node 20 (Leaf 10) (Leaf 12)))
             (Node 10 (Leaf 12) (Node 20 (Leaf 15) (Leaf 18))))
```

(7 Marks)

6. Consider a variation of the language done in the class:

```
data Expr = I Var | PP Var | Add Expr Expr | Sub Expr Expr
          | If BExpr Expr Expr | Pp Var | Label String Expr
data BExpr = Gt Expr Expr | Not BExpr
data Var = X | Y | Z
```

Write a evaluator for the language called `eval`, which, apart from the usual interpretation, will also give profiling information as the number of times each labelled expression is evaluated[1]. This information is to be maintained in a list such as: `[("L1",0), ("L2",1), ("L3",1)]`. Use the `do` notation and the get and the set functions. Also name the initial state as `initState` and provide a function called `showState`, which will convert the state to a string form. Thus one should be able to say: `showState (snd (eval e initState))` to see the final state. (13 Marks)

getting state correct :- 2

sub, add → 1

~~Sub~~ → 1

Not & Gt → 1 ~~Not → 1~~

if → 3 Label → 2

update state → 2

non eval → 2

2 + 1 + 1 + 1

2 +

[1] For our language this number is 0 or 1. However, if we add functions to this language, we may get numbers other than 0 or 1.