Homework 4

Due 22 October (ungraded)

Handout 5
CS242: Autumn 2008
15 October

Reading

1. Read "Tackling the Awkward Squad," Sections 1-2
2. Read "Monads for functional programming," Section 1-3
3. Read "A History of Haskell: Being lazy with Class," Section 6.4 and Section 7

   See `http://www.stanford.edu/class/cs242/readings/` for links to these readings.

Problems

1. .... Tree numbering using Monads

Consider the binary tree datatype as describe in homework 2.

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

The following declaration of a state monad is described after the code:

```
newtype State s a = State { runState :: s -> (a,s)}
-- runState :: State s a -> s -> (a,s)
-- This function simply unwraps a value of type State
-- Doing this unwrapping and applying the resulting function
-- has the effect of "running" the monad.

returnState :: a -> State s a
returnState a = State (\s -> (a,s))

bindState :: State s a -> (a -> State s b) -> State s b
bindState m k = State (
          \oldState ->
          let (a,s') = runState m oldState
          in   runState (k a) s' )

instance Monad (State s) where
  return = returnState
  (>>=) = bindState

get :: State s s
get = State (\s -> (s,s))

put :: s -> State s ()
put s = State (\s' -> ((),s))

-- The update function updates the  state by f and returns the old value of state.
update :: (s->s) -> State s s
update f = State (\s -> (s,f s))
```

Explanation of the above code : To make State an instance of the typeclass monad (enabling the do notation), we need to have a type constructor for the monad. The declaration

$$\texttt{newtype State s a} = \texttt{State} \{\texttt{runState} :: \texttt{s} \rightarrow (\texttt{a}, \texttt{s})\}$$

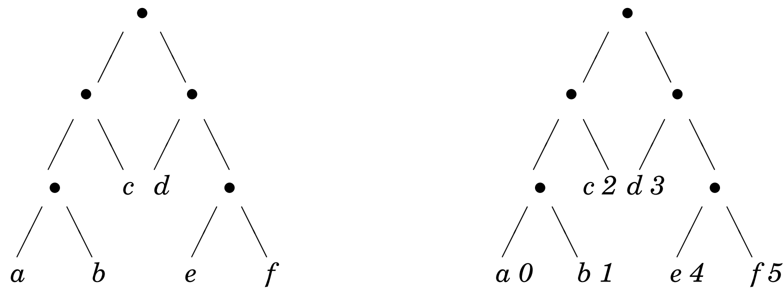
introduces a new type with constructor `State` and deconstructor `runState`. To build a value of this type, we apply the constructor `State` to a function from a state to a pair of a value and a state. To unwrap a value of this type, we apply the function `runState`, which has the type:

$$\texttt{runState} :: \texttt{State s a} \rightarrow \texttt{s} \rightarrow (\texttt{a}, \texttt{s})$$

When `runState` is applied to an element `m` of type `State s a`, it returns the argument that was supplied to `m`, that is, a function of type $\texttt{s} \rightarrow (\texttt{a}, \texttt{s})$.

The declaration "`instance Monad (State s) where`..." makes State an instance of the type class Monad and specifies the `return` and >>= (bind) functions. The operations `get`, `put` and `update` manipulate the state. The `get` operation simply returns the state, while `put` simply updates it and returns the unit value. The `update` function returns the current state, like `get`. In the process though, it updates the state according to its argument function `f`, storing the result, so the next time the state is accessed, it will have been modified by `f`.

Answer the following questions:



(a) Write a non-monadic function $\texttt{numberLeaves} :: \texttt{Tree a} \rightarrow \texttt{Tree (a, Int)}$ which takes a Tree as argument and return a new tree numbered from left to right as shown in the figure above.

(b) Write the same function monadically, given the template below.

```
numberLeavesM :: Tree a -> Tree (a,Int)
numberLeavesM t = fst(runState (number t) 0)
  where
    number (Leaf i) =  < ... >
    number (Node l r) = < ... >
```

The type of the `number` function in the above template is $\texttt{number} :: \texttt{Tree t} \rightarrow \texttt{State Int (Tree(t, Int))}$.
Hint : Number the nodes recursively. Use the state to keep track of the numbering and the value to keep track of the numbered tree. Use the update function to increment the state each time a leaf is numbered.

(c) Modify the non-monadic version to number the nodes from right to left.

(d) Modify the monadic version to number the nodes from right to left

(e) Which version was easier to modify? Consider if the tree had had more than just two children.

2. .... Custom Haskell Control Structures

One of the claimed advantages of first-class actions is the ability to write custom control structures. This question will explore that ability by asking you to write some familiar control structures.

Below, we provide code for the `whileIO` control structure implemented in the IO monad. We also provide an example of its usage which prints the integers between 0 and 3.

```
import Control.Monad.ST
import Data.IORef

whileIO :: IO Bool -> IO () -> IO ()
whileIO b m  = ifIO b
                (do {m; whileIO b m})
                (return ())

whileTest = do
             {v <- newIORef 0;
              whileIO (do{ x <- readIORef v;
                           return (x<4)})
                      (do{ x<-readIORef v;
                           print x;
                           writeIORef v (1+x) })}
```

(a) The specification for `whileIO` above is not complete without the `ifIO` custom control structure. Please implement the appropriate `ifIO` action. We provide the type signature and an example usage of the action below.

```
ifIO :: IO Bool -> IO a -> IO a -> IO a

ifTest = ifIO (return (3<4)) (print "True") (print "False")
```

(b) `untilIO` is a very similar control structure to `whileIO`, except the loop condition test 1) executes after the loop body and 2) causes loop exit if `true` instead of `false`, as is the case for `whileIO`. Please implement `untilIO` with the provided type signature and also create `untilTest` to print the integers between 0 and 3.

```
untilIO:: IO () -> IO Bool -> IO ()
```

3. .... Monads in Haskell

In Haskell, `(>>)` and `done` can be defined as follows:

```
(>>) :: IO a -> IO b -> IO b
m >> n = m >>= (\_ -> n)

done:: IO ()
done = return ()
```

with the following derived monad laws

```
         done >> m   =  m
         m >> done   =  m
  m1 >> (m2 >> m3)   =  (m1 >> m2) >> m3
```

Now consider the following implementations of `putStr` and `(++)`:

```
-- outputs a string using the IO monad
putStr :: String -> IO ()
putStr [] = done
putStr (c:cs) = putChar c >> putStr cs

-- concatenates two strings
(++) :: String -> String -> String
(++) [] s = s
(++) (c:cs) s = c : (cs ++ s)
```

Given the above definitions and derived monad laws, prove

```
putStr x >> putStr y = putStr (x ++ y)
```

(Hint: You should use induction on the form of `x`)

4. .... Exiting the IO Monad

Tackling the Awkward Squad explains how Haskell uses monads to perform IO while remaining in the functional paradigm. However, occasionally, we wish to exit the IO monad. We can do this through the method

```
unsafePerformIO :: IO a -> a
```

which extracts a value of type `a` from the monadic type `IO a`.

(a) Explain why `unsafePerformIO` is dangerous, even assuming that you are not using it to break the type system. Give an example where its use leads to unpredictable behavior.

(b) Under what circumstances is it appropriate to use `unsafePerformIO`? Give three different example situations and explain why each is an acceptable use.