1 Introduction

This document is a derivative of many sources, most notably (in no particular order):

5. Materials of TDA555 from Chalmers,
   http://www.cse.chalmers.se/edu/course/TDA555/.

2 Exercises

2.1 Simple definitions (Thompson)

Show that binary trees can be given a monad structure. Show the same for the type

\[
\text{data Error } a = \text{OK } a \mid \text{Error String}
\]

2.2 Monadic helper functions (TDA555)

Give an implementation of the following two functions:

\[
\text{sequence} :: \text{Monad } m \Rightarrow [m ()] \rightarrow m ()
\]

\[
\text{onlyIf} :: \text{Monad } m \Rightarrow \text{Bool } \rightarrow m () \rightarrow m ()
\]

\text{sequence} takes a list of instructions resulting in an uninteresting value, and creates one big instruction that executes all of these.
\text{onlyIf} takes a boolean and an instruction, and creates an instruction that only executes the argument instruction if the boolean was \text{True}. If the boolean was \text{False}, the instruction is an uninteresting value.

*Intended for EDAN40 course, after the monad lecture.
was False, nothing happens. Example onlyIf failed tryAgain executes the
instructions tryAgain only if the boolean failed is True.

**Hint:** You might find it easier to think of the above functions having type:

```haskell
sequence_ :: [IO ()] -> IO ()
onlyIf :: Bool -> IO () -> IO ()
```

What becomes different if we change the type of onlyIf to:

```haskell
onlyIfM :: Monad m => m Bool -> m () -> m ()
```

What other kinds of programs can we write now?

Give an implementation of onlyIfM.

### 2.3 List comprehension (lecture)

Let us take the example from the lecture:

```haskell
list1 = [ (x,y) | x<-[1..], y<-[1..x] ]
list2 = do
  x <- [1..]
  y <- [1..x]
  return (x,y)
```

Rewrite `list2` using bind (>>=) instead of do.

### 2.4 Some theorem proving (Thompson)

First, let us remind that the full Monad class is defined as follows:

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>) :: m a -> m b -> m b
  fail :: String -> m a

where (>>) and fail may be defined as defaults:

```haskell
m >>= k = m >>= \_ -> k
fail s = error s
```

So >>= just discards the value of the first expression, whereas >>= passes it to
the next one.

Let us now define yet another derived operator, >@>:

```haskell
(@>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >@> g = \x -> (f x) >>= g
```

This is the *Kleisli composition.*

Let us now formulate three requirements on the operations of a monad. First, return acts as identity for the operator >@>:

2
\[
\begin{align*}
\text{return} \circ \circ f &= f & \text{(M1)} \\
\ circ \ circ \text{return} &= \ circ f & \text{(M2)}
\end{align*}
\]
and the operator \(\circ\circ\) should be associative:
\[
(f \circ \circ g) \circ \circ h = f \circ \circ (g \circ \circ h) & \text{ (M3)}
\]

**Task 1:** For the monads \(\text{Id}, [], \text{Maybe}\) prove the rules (M1) to (M3).

Let us define some standard functions over every monad:

\[
\begin{align*}
\text{fmap} & \colon \text{Monad } m \Rightarrow (a \to b) \to m a \to m b \\
\text{join} & \colon \text{Monad } m \Rightarrow m (m a) \to m a
\end{align*}
\]

\[
\begin{align*}
\text{fmap } f \ m &= \text{do } x \leftarrow m \\
&\hspace{1em} \text{return } (f \ x) \\
\text{join } m &= \text{do } x \leftarrow m \\
&\hspace{1em} x
\end{align*}
\]

Over the lists they correspond to \text{map} and \text{concat}, respectively. Now we can formulate the following dependency:

\[
\text{fmap } (f \circ g) = \text{fmap } f \circ \text{fmap } g & \text{ (M4)}
\]

**Task 2:** Prove the property (M4) using the laws (M1) to (M3).

**Task 3:** Prove the following properties using the monad laws:

\[
\begin{align*}
\text{join return} &= \text{join } \circ \text{fmap } \text{return} \\
\text{join return} &= \text{id}
\end{align*}
\]

**Task 4:** Write down the definitions of \text{map} and \text{join} over lists using list comprehensions. Compare them with the definitions of \text{fmap} and \text{join} given in the do-notation above.

### 2.5 State Monad (RealWorldHaskell)

You may wish to have a look at Chapter 14 of “Real World Haskell”\(^1\) where, among others, the State monad is gently introduced. In particular, the following set of definitions come there:

\[
\begin{align*}
\text{newtype State } s \ a &= \text{State } \{ \\
&\hspace{1em} \text{runState} : s \to (a, s) \\
\} \\
\text{returnState} &\colon a \to \text{State } s \ a \\
\text{returnState } a &= \text{State } \$ \ s \to (a, s) \\
\text{bindState} &\colon \text{State } s \ a \to (a \to \text{State } s \ b) \to \text{State } s \ b \\
\text{bindState } m \ k &= \text{State } \$ \ s \to \text{let } (a, s') = \text{runState } m \ s \\
&\hspace{1em} \text{in } \text{runState } (k \ a) \ s'
\end{align*}
\]

\(^1\)http://book.realworldhaskell.org
instance Monad (State s) where
    return a = returnState a
    m >>= k = bindState m k

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

put :: s -> State s ()
put s = State $ \_ -> ((), s)

**Task 1:** (and only one:-) Rewrite the random number generator as an instance of the state monad, using the functions **put** and **get**.

The *StateTransform* monad from the lecture notes is actually equivalent to the above *State* monad, with possibly a slightly different set of help functions defined, in particular:

updateST :: (s -> s) -> State s ()
updateST u = State $ \s -> ((), u s)

queryST :: (s -> a) -> State s a
queryST q = State $ \s -> (q s, s)

Finally, please note that the *StateTransform* monad is **not** a monad transformer (as defined in Chapter 18 of RWH).