DevanshR

(SIGNATURE)

FUNCTIONAL PROGRAMMING - CS5502

End-Semester Answer Sheet

NAME: *Devansh Singh Rathore*
ROLL NO.: *111701011*
SEMESTER: *6*
INSTITUTE: *Indian Institute of Technology, Palakkad*

>> Question 1 >>

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = pure []
sequence (ma:mas) = do a <- ma
                       as <- sequence mas
                       return (a:as)
```

>> Question 2 >>

```
foo act = (+) <$> act <*> act

{-
     Explanation:

     Earlier for, foo act = do x <- act
                               xs <- act
                               pure (x + xs)
     foo :: (Monad m, Int a) => m a -> m a

     Now,
     foo :: (Applicative m, Int a) => m a -> m a
-}
```

>> Question 3 >>

```
newtype State s a = State { runState :: s -> (a,s) }
```

```
-- Applicative instance:
instance Applicative (State s) where
pure :: a -> ((State s) a)
pure a = State t where t s2 = (a, s2)

(<*>) :: State s (x -> y) -> (State s) x -> (State s) y
(<*>) sx sy = State t
              where t s2 = let
                              (f, h) = runState sx s2
                              (a, k) = runState sy h
                           in (f a, k)
```

>> Question 4 >>

```
data Snack = Samosa | Vada

-- The user input
data Input = Money Int | Demand Snack | Change

-- This action dispenses a given snack to the user.
snack :: Snack -- the snack to be given out.
         -> IO ()

-- This action pays a given amount of money to the user.
-- Can be used to handle the change action.
pay :: Int      -- Give out so-much rupee.
       -> IO ()

-- Wait for the next user input and return it.
getUserInput :: IO Input

-- Use to display some message to the user. Info/Error etc
display :: String    -- The message to be displayed
           -> IO ()
```

(a) `type VendM a = StateT Int IO a`

(SIGNATURE)

(b)
```
import Control.Monad.State (get, put)
get :: VendM Int
put :: Int -> VendM ()

io :: IO a -> VendM a
io = liftIO

cost :: Snack -> Int
cost Samosa = 8
cost Vada = 7

increaseBalance :: Int -> VendM()
increaseBalance x = do balance <- get
                    put $ balance + x

change :: VendM ()
change = do balance <- get
            io $ pay balance
            put 0
```

(c)
```
serve :: Snack -> VendM ()
serve snk = do balance <- get
               fn snk
            where fn snk = let p = cost snk
                           in
                            if p > balance
                            then io $ display "Error (less
                        balance)"
                            else io $ Snack snk >> put
                        (balance - p)
```

(d)
```
vend :: VendM ()
vend = do input <- getUserInput
          operation
          vend
       where operation = case input of
                             Money b -> increaseBalance b
                             Demand snk -> serve snk
                             Change -> change
```

(SIGNATURE)

(e)

```
main :: IO ()
main = runStateT vend 0 >> return ()
```