1. Function Type Declarations (12 points):
For each of the following function definitions, correctly complete the preceding type declaration. Be sure to include any necessary class constraints.

(A)  
\[
\text{mystery1} :: (a \to b \to c) \to [a] \to [b] \to [c] \\
mystery1 \ g \ [] \ = \ [] \\
mystery1 \ g \ [] \ _\ = \ [] \\
mystery1 \ g \ (x:xs) \ (y:ys) \ = \ g \ x \ y : \ mystery1 \ g \ xs \ ys
\]

(B)  
\[
\text{mystery2} :: [(a \to b \to c)] \to a \to b \to [c] \\
mystery2 \ gs \ x \ y \ = \ \text{map} \ (\lambda h \to h \ y) \ \text{map} \ (\lambda g \to g \ x) \ gs
\]

(C)  
\[
\text{mystery3} :: (\text{Applicative} \ a, \text{Ord} \ b) \Rightarrow a \ b \to a \ b \to a \ b \\
mystery3 \ x \ y \ = \ \text{pure} \ \text{max} \ (\leftrightarrow) \ x \ (\leftrightarrow) \ y
\]

(D)  
\[
\text{mystery4} :: (\text{Monad} \ m) \Rightarrow m \ a \to (a \to m \ b) \to (a \to b \to c) \to m \ c \\
mystery4 \ x \ f \ g \ = \ \text{do} \ \ n \ <- \ f \ m \\
\text{return} \ $ \ g \ m \ n
\]
2. Defining Functors, Applicatives, and Monads (12 points):

Consider the following data type:

```haskell
data Box a = Gift a | ReGift (Box a) deriving Show
```

The `Box` type can be used to keep track of the contents of a gift box, and additionally reflect how many times the contents have been unpacked and “re-gifted”. E.g.,

```haskell
eg_box_1 = Gift "A brand new sweater"
eg_box_2 = ReGift (Gift "A slightly used sweater")
eg_box_3 = ReGift (ReGift (ReGift (Gift "A much used sweater")))
```

On the next page you are to implement the `Functor`, `Applicative`, and `Monad` typeclass instances for the `Box` type. The `Applicative` and `Monad` functions will automatically “wrap” `Boxes` in additional layers of `ReGift` containers as they are combined together and sequenced.

The following examples show the `fmap`, `<*>`, and `>>=` operators in action, along with their results (in comments):

```haskell
fmap ("New "++) (Gift "Jeans")
--=> Gift "New Jeans"

fmap ("Used "++) (ReGift (ReGift (Gift "Jeans")))
--=> ReGift (ReGift (Gift "Used Jeans"))

Gift ("T-Shirt and "++) <*> Gift ("Jeans")
--=> ReGift (ReGift (Gift "T-Shirt and Jeans"))

ReGift (Gift ("T-Shirt and "++)) <*> ReGift (Gift ("Jeans"))
--=> ReGift (ReGift (ReGift (Gift "T-Shirt and Jeans"))))

do g <- Gift "Jeans"
   return g
--=> ReGift (Gift "Jeans")

do g1 <- Gift "Jeans"
g2 <- Gift ("New-ish " ++ g1)
g3 <- Gift ("Sorta " ++ g2)
g4 <- Gift ("Kinda " ++ g3)
   return g4
--=> ReGift (ReGift (ReGift (Gift "Kinda Sorta New-ish Jeans"))))
```
instance Functor Box where
    -- fmap :: (a -> b) -> Box a -> Box b
    fmap f (Gift x) = Gift $ f x
    fmap f (ReGift b) = ReGift $ fmap f b

instance Applicative Box where
    pure x = Gift x
    -- ( <*> ) :: Box (a -> b) -> Box a -> Box b
    Gift f <*> Gift x = ReGift $ ReGift $ Gift $ f x
    Gift f <*> ReGift b = ReGift $ Gift f <*> b
    ReGift b <*> c = ReGift $ b <*> c

instance Monad Box where
    return = pure
    -- ( >>= ) :: Box a -> (a -> Box b) -> Box b
    Gift x >>= f = ReGift $ f x
    ReGift b >>= f = ReGift $ b >>= f
3. Using the State Monad (16 points):

Consider the following functions that return State monads.

```haskell
scroll :: Int -> State [a] a
scroll n = State $ \xs -> let n' = if n >= 0 then n else length xs + n
          in (head ns, ns)

put :: a -> State [a] a
put x' = State $ \(x:xs) -> (x',x':xs)

alter :: (a -> a) -> State [a] a
alter f = State $ \(x:xs) -> let y = f x in (y,y:xs)
```

For each of the following, determine the return value of the call to `runState`. Note that the definition of the State monad is provided at the end of the exam.

(A) `runState (put 55) [1..10]`  
   ```haskell
   --=> (55,[55,2,3,4,5,6,7,8,9,10])
   ```

(B) `runState (pure (\x y -> (x,y)) <*> scroll 3 <*> alter (3*)) [1..10]`  
   ```haskell
   --=> ((4,12),[12,5,6,7,8,9,10,1,2,3])
   ```

(C) `sC = do
scroll 2
alter reverse
runState sC ["hello", "hola", "aloha", "bonjour"]`  
   ```haskell
   --=> ("ahola",["ahola","bonjour","hello","hola"])```

(D) `sD = do
a <- scroll 1
scroll 2
b <- alter (+a)
c <- scroll 4
alter (*b)
scroll (-3)
put c
runState sD [1..10]`  
   ```haskell
   --=> (8,[8,6,7,48,9,10,1,2,3,6])```
4. Monadic Parsing (12 points):

Consider the following grammar for a simple language for looping and printing:

\[
\begin{align*}
\text{prog} &::= \text{block} \\
\text{block} &::= \text{BEGIN} \text{ statement* END} \\
\text{statement} &::= \text{loop stmt} \mid \text{print stmt} \\
\text{loop stmt} &::= \text{LOOP} \text{ natural (statement} \mid \text{block) } \\
\text{print stmt} &::= \text{PRINT} \text{ string} \\
\end{align*}
\]

I.e., a program (\text{prog}) is a \text{block} of zero or more \text{statements} enclosed within \text{BEGIN} and \text{END} tokens. Each \text{statement} is either a \text{loop stmt} (starting with \text{LOOP} followed by a \text{natural} number then by a \text{statement} or \text{block}), or a \text{print stmt} (starting with \text{PRINT} and followed by a \text{string}).

The following are some sample programs that adhere to this grammar:

\[
\begin{align*}
\text{BEGIN} \\
&\quad \text{PRINT} \ "hello world" \\
\text{END} \\
&\text{BEGIN} \\
&\quad \text{LOOP} 2 \\
&\quad \text{BEGIN} \\
&\quad&\quad \text{PRINT} \ "hello" \\
&\quad&\quad \text{PRINT} \ "world" \\
&\quad \text{END} \\
&\text{END} \\
&\text{BEGIN} \\
&\quad \text{LOOP} 10 \\
&\quad \text{BEGIN} \\
&\quad&\quad \text{PRINT} \ "1" \\
&\quad&\quad \text{LOOP} 20 \\
&\quad&\quad \text{LOOP} 30 \\
&\quad&\quad \text{PRINT} \ "2" \\
&\quad \text{END} \\
&\quad \text{PRINT} \ "3" \\
&\quad \text{LOOP} 40 \\
&\quad \text{PRINT} \ "4" \\
&\text{END}
\end{align*}
\]

On the next page, implement \text{prog}, which is a parser for programs as specified above. You may define as many other parsers as you wish to call from \text{prog}. The \text{Parser} monad and related functions are given at the end of the exam — note that we have additionally provided the \text{quotedString} parser, which will correctly parse double-quote enclosed characters.

Note that your implementation need only successfully parse input strings that conform to the above grammar (and fail otherwise). You do \textbf{not} need to evaluate the input string in any other way.
prog :: Parser ()
    prog = block

    block :: Parser ()
    block = do symbol "BEGIN"
              many statement
              symbol "END"
              return ()

statement :: Parser ()
    statement = loop_stmt <|> print_stmt

    loop_stmt :: Parser ()
    loop_stmt = do symbol "LOOP"
                 natural
                 statement <|> block
                 return ()

print_stmt :: Parser ()
    print_stmt = do symbol "PRINT"
                 quotedString
                 return ()