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)

```
mystery1 :: (a -> b -> c) -> [a] -> [b] -> [c]

mystery1 g _ [] = []
mystery1 g [] _ = []
mystery1 g (x:xs) (y:ys) = g x y : mystery1 g xs ys
```

(B)

```
mystery2 :: [(a -> b -> c)] -> a -> b -> [c]

mystery2 gs x y = map (\h -> h y)  $ map (\g -> g x) gs
```

(C)

```
mystery3 :: (Applicative a, Ord b) => a b -> a b -> a b

mystery3 x y = pure max <*> x <*> y
```

(D)

```
mystery4 :: (Monad m) => m a -> (a -> m b) -> (a -> b -> c) -> m c

mystery4 x f g = do m <- x
                    n <- f m
                    return $ g m n
```

2. Defining Functors, Applicatives, and Monads (12 points):

Consider the following data type:

```
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.,

```
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" `Box`es 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):

```
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 (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 (ReGift (Gift "Kinda Sorta New-ish Jeans"))))
```

```haskell
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.

```
scroll :: Int -> State [a] a
scroll n = State $ \xs -> let n' = if n >= 0 then n else length xs + n
                              ns = (drop n' xs) ++ (take n' xs)
                          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]
--=> (55,[55,2,3,4,5,6,7,8,9,10])
```

(B)

```
runState (pure (\x y -> (x,y)) <*> scroll 3 <*> alter (3*)) [1..10]
--=> ((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"]
--=> ("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]
--=> (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:

> *prog ::= block*
> *block ::= BEGIN statement* END*
> *statement ::= loop_stmt | print_stmt*
> *loop_stmt ::= LOOP natural (statement | block)*
> *print_stmt ::= PRINT string*

I.e., a program (*prog*) is a *block* of zero or more *statement*s enclosed within `BEGIN` and `END` tokens. Each *statement* is either a *loop_stmt* (starting with `LOOP` followed by a *natural* number then by a *statement* or *block*), or a *print_stmt* (starting with `PRINT` and followed by a *string*).

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

```
BEGIN
  PRINT "hello world"
END

BEGIN
  LOOP 2
  BEGIN
    PRINT "hello"
    PRINT "world"
  END
END

BEGIN
  LOOP 10
  BEGIN
    PRINT "1"
    LOOP 20
      LOOP 30
        PRINT "2"
  END
  PRINT "3"
  LOOP 40
    PRINT "4"
END
```

On the next page, implement `prog`, which is a parser for programs as specified above. You may define as many other parsers as you wish to call from `prog`. The `Parser` monad and related functions are given at the end of the exam — note that we have additionally provided the `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 **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 ()
```