CS 495 Spring 2018

Final Exam

Instructions:

- This exam is closed-book, closed-notes. Computers of any kind are not permitted.
- Write your final answers, tidily, in the boxes provided. Separate scratch paper will be provided — please ask if you need more.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1 | (/12) :
| 2 | (/12) :
| 3 | (/16) :
| 4 | (/12) :
| TOTAL | (/52) : |
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 ::

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

(B)

mystery2 ::

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

(C)

mystery3 ::

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

(D)

mystery4 ::

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:

```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” `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):

```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

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

instance Monad Box where
    return = pure
    -- (>>=) :: Box a -> (a -> Box b) -> Box b
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 \texttt{runState}. Note that the definition of the \texttt{State} monad is provided at the end of the exam.

(A) \[
\text{runState (put 55) [1..10]}
\]

(B) \[
\text{runState (pure (\ x y -> (x,y)) <*> scroll 3 <*> alter (3*)) [1..10]}
\]
(C) \[
\begin{align*}
\text{sC} = \text{do} \\
& \quad \text{scroll 2} \\
& \quad \text{alter reverse} \\
\text{runState sC ["hello", "hola", "aloha", "bonjour"]}
\end{align*}
\]

(D) \[
\begin{align*}
\text{sD} = \text{do} \\
& \quad a \leftarrow \text{scroll 1} \\
& \quad \text{scroll 2} \\
& \quad b \leftarrow \text{alter (+a)} \\
& \quad c \leftarrow \text{scroll 4} \\
& \quad \text{alter (*b)} \\
& \quad \text{scroll (-3)} \\
& \quad \text{put c}
\end{align*}
\]

\[
\text{runState sD [1..10]}
\]
4. Monadic Parsing (12 points):

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

\[
\begin{align*}
\text{prog} & ::= \text{block} \\
\text{block} & ::= \text{BEGIN statement* END} \\
\text{statement} & ::= \text{loop_stmt | print_stmt} \\
\text{loop_stmt} & ::= \text{LOOP natural (statement | block)} \\
\text{print_stmt} & ::= \text{PRINT string}
\end{align*}
\]

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:

\[
\begin{align*}
\text{BEGIN} & \\
& \quad \text{PRINT } "hello world" \\
\text{END}
\end{align*}
\]

\[
\begin{align*}
\text{BEGIN} & \\
& \quad \text{LOOP 2} \\
& \quad \text{BEGIN} \\
& \quad \quad \text{PRINT } "hello" \\
& \quad \quad \text{PRINT } "world" \\
& \quad \text{END} \\
\text{END}
\end{align*}
\]

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

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 =
Source Listing

-- State Monad

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

instance Functor (State s) where
    fmap f st = State $ \s -> let (x, s') = runState st s
        in (f x, s')

instance Applicative (State s) where
    pure x = State $ \s -> (x, s)
    stf <*> stx = State $ \s -> let (f, s') = runState stf s
                             (x, s'') = runState stx s'
                              in (f x, s'')

instance Monad (State s) where
    return x = State $ \s -> (x, s)
    st >>= f = State $ \s -> let (x, s') = runState st s
                             in runState (f x) s'
```

-- Parser Monad

```haskell
data Parser a = Parser { parse :: String -> [(a, String)] }

item :: Parser Char
item = Parser $ \inp -> case inp of [] -> []
    (x:xs) -> [(x, xs)]

instance Functor Parser where
    fmap f p = Parser $ \inp -> case parse p inp of
        [] -> []
        [(v, out)] -> [(f v, out)]

instance Applicative Parser where
    pure v = Parser $ \inp -> [(v, inp)]
    pf <*> px = Parser $ \inp -> case parse pf inp of
        [] -> []
        [(f, out)] -> parse (fmap f px) out

instance Monad Parser where
    p >>= f = Parser $ \inp -> case parse p inp of
        [] -> []
        [(v, out)] -> parse (f v) out

instance Alternative Parser where
    empty = Parser $ \inp -> []
    p <|> q = Parser $ \inp -> case parse p inp of
        [] -> parse q inp
        res -> res

sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
          if p x then return x else empty

digit :: Parser Char
digit = sat isDigit
```
lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char -> Parser Char
char x = sat (== x)

string :: String -> Parser String
string "" = return ""
string (x:xs) = do char x
                      string xs
                      return (x:xs)

ident :: Parser String
ident = do x <- lower
           xs <- many alphanum
           return (x:xs)

nat :: Parser Int
nat = do xs <- some digit
        return ( read xs)

space :: Parser ()
space = do many ( sat isSpace)
        return ()

quoted :: Parser String
quoted = do char '\''
           s <- many ( sat (/='\''))
           char '\''
           return s

token :: Parser a -> Parser a
token p = do space
            v <- p
            space
            return v

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

symbol :: String -> Parser String
symbol xs = token (string xs)

quotedString :: Parser String
quotedString = token quoted