1. Evaluating Folds (9 points):
Show the result of evaluating each of the following expressions involving either foldr or foldl.

(A) \[
\text{foldl } (\langle y_1, y_2 \rangle \ x \rightarrow ((\text{min } x \ y_1), \ (\text{max } x \ y_2))) \\
(100,0) \\
[5,10..95] \\
> (5, 95)
\]

(B) \[
\text{foldr } \text{iter } ["] \ "\text{hello}" \\
\text{where } \text{iter } x \ y = \text{map } (x:) ("":y) \\
> ["h","he","hel","hell","hello","hello"]
\]

(C) \[
\text{foldl } \text{iter } \text{id } ["ad", \ "id", \ "al"] \ "CS" \\
\text{where } \text{iter } r \ x = \ \w -> r (\w ++ "-" ++ \text{reverse } x) \\
> "CS-la-di-da"
\]
2. Defining Functors, Applicatives, and Monads (12 points):

Consider the following data type:

```haskell
data Weighted a = WZero a | WVar Int a
```

The `Weighted` type is used to assign arbitrary integer weights to values (`WZero` implies a weight of 0, while `WVar` lets us attach an `Int` weight). When `Weighted` values are combined via `Applicative` or `Monad` functions, weights are simply summed.

Examples of using `Weighted` values as Functors, Applicatives, and Monads follow:

```
fmap reverse (WVar 5 "hello")    -- > WVar 5 "olleh"
pure reverse <*> pure "hello"     -- > WZero "olleh"
(++) <$> WVar 2 "hello" <*> WVar 3 "world" -- > WVar 5 "helloworld"
do v1 <- WVar 3 "This"
v2 <- WZero "Is"
v3 <- WVar 7 "An"
v4 <- WZero "Example"
return (v1++v2++v3++v4)            -- > WVar 10 "ThisIsAnExample"
```

Define the `Weighted` Functor, Applicative, and Monad instances on the next page.

```haskell
instance Functor Weighted where
  fmap f (WZero x) = WZero $ f x
  fmap f (WVar n x) = WVar n $ f x

instance Applicative Weighted where
  pure x = WZero x
  (WZero f) <*> (WZero x) = WZero $ f x
  (WZero f) <*> (WVar n x) = WVar n $ f x
  (WVar n f) <*> (WZero x) = WVar n $ f x
  (WVar m f) <*> (WVar n x) = WVar (m+n) (f x)

instance Monad Weighted where
  return = pure
  (WZero x) >>= f = f x
  (WVar m x) >>= f = case f x of (WZero y) -> WVar m y
                               (WVar n y) -> WVar (m+n) y
```

Page 2 of 7
3. Using the State Monad (12 points):

Consider the following functions that return \texttt{State} monads.

\begin{verbatim}
  fwd :: Int \to \texttt{State} [a] ()
fwd n = State \$ \lambda xs \to (((), \text{drop } n \text{ } xs ++ \text{take } n \text{ } xs)

  rew :: Int \to \texttt{State} [a] ()
rew n = State \$ \lambda xs \to \text{let } n' = \text{length } xs - n
  in (((), \text{drop } n' \text{ } xs ++ \text{take } n' \text{ } xs)

  swp :: a \to \texttt{State} [a] a
swp x = State \$ \lambda(y:ys) \to (y, x:ys)

  red :: (a \to a \to a) \to \texttt{State} [a] a
red f = State \$ \lambda l@(x:xs) \to (\text{foldr } f x xs, l)
\end{verbatim}

For each of the following, determine the return value of the call to \texttt{run}. Note that the definition of the \texttt{State} monad is provided at the end of the exam.

(A) \begin{verbatim}
run (fwd 3) [1..10]
\end{verbatim}
\[
> ((),[4,5,6,7,8,9,10,1,2,3])
\]

(B) \begin{verbatim}
run (fmap (+50) (swp 8)) [1..10]
\end{verbatim}
\[
> (51,[8,2,3,4,5,6,7,8,9,10])
\]

(C) \begin{verbatim}
run (pure (*) <*> (swp 10) <*> (red (-))) [2, 4, 7]
\end{verbatim}
\[
> (-2,[10,4,7])
\]

(D) \begin{verbatim}
run (do x <- swp "the"
  fwd 1
  y <- swp "red"
  rew 4
  fwd 3) ["smurfs", "are", "small", "and", "blue"]
\end{verbatim}
\[
> ((),["the","red","small","and","blue"])
\]
4. Monadic Parsing (12 points):

For this problem you are to implement a monadic parser for a simple subset of HTML, where valid input consists of a properly formatted element, identified by matching opening and closing tags of the form `<TAGNAME>` and `</TAGNAME>`. Tag names can be made up of only alphabetical characters (lower and uppercase). An element can contain zero or more elements, and elements can also be nested.

After parsing valid input, your parser should return the names of the elements in a tree that mimics the structure of the input, where the tree data type is defined as follows:

```haskell
data Tree a = Node a [Tree a]
```

Below are sample valid inputs, each accompanied by the tree obtained by parsing it (note that indentation is not important to the syntax):

```
<foo></foo>
-- > Node "foo" []

<a>
  <b>
    <c></c>
  </b>
</a>
-- > Node "a" [Node "b" [Node "c" []]]

<a>
  <b></b>
  <c></c>
</a>
-- > Node "a" [Node "b" [],Node "c" []]
```

On the next page, implement the parser `element`. You may define as many other parsers as you wish to use in `element`. The `Parser` monad and related functions are given at the end of the exam.

A parser that succeeds on valid input and fails on invalid input will receive 75% of the points; additionally returning a correct tree will earn full points.
element :: Parser (Tree String)
element = do name <- openTag
              names <- many element
              closeTag name
              return (Node name names)

tagName :: Parser String
tagName = some alpha

openTag :: Parser String
openTag = do symbol "<"
            name <- tagName
            symbol ">
            return name

closeTag :: String -> Parser ()
closeTag name = do symbol "</"
                  symbol name
                  symbol ">
                  return ()
5. Evaluating Search (12 points):

For this problem we’ll consider a simple type of “sliding pieces” puzzle, which consists of 2 or more rows of values. The values in each row can be shifted to the left to change their ordering. The puzzle is considered solved when the values across all rows have the same ordering.

E.g., the puzzle \[\text{["ABCD", "DABC", "ABCD"]}\] has three rows of values, each row containing 4 characters. To solve this puzzle, we could shift the row "DABC" once, which takes the 'D' from the front and moves it to the end of the row — the resulting row, "ABCD", matches the others, and so we are done.

Below we define types and a function used to represent such puzzles and to try out moves. Each entry in a move list corresponds to the index of a row to be shifted once.

```haskell
type Puzzle = [String]
type PuzzleMoves = [Int]
runMoves :: Puzzle -> PuzzleMoves -> Puzzle
runMoves p sol = foldl flipPuzz p sol
  where flipPuzz p n = let row = p !! n
                        row' = drop 1 row ++ take 1 row
                        in take n p ++ [row'] ++ drop (n+1) p

The following are sample calls to runMoves, along with their results (illustrating three different ways of solving the puzzle described above):

runMoves ["ABCD", "DABC", "ABCD"] [0,2,1,1] -- > ["BCDA", "BCDA", "BCDA"]
runMoves ["ABCD", "DABC", "ABCD"] [0,0,0,2,2,2] -- > ["DABC", "DABC", "DABC"]

The (partly defined) function puzzleSearch searches for a solution to a provided puzzle using bestFirstSearch (given at the end of the exam). Answer the questions on the following page based on puzzleSearch.

```
(A) Implement a suitable goal function for `puzzleSearch`. (You may use `runMoves` in your implementation.)

```haskell
  goal sol = let sol' = runMoves puzz sol
               in all (== head sol') (tail sol')
```

(B) Implement a score function for `puzzleSearch` which will enable it to find a solution that takes a minimal number of moves.

```haskell
  score sol = length sol -- (minimal solution)
```

(C) Consider the following definition of score:

```haskell
  score sol = let puzz' = runMoves puzz sol
               in length $ filter (/= head puzz') (tail puzz')
```

Assuming that the goal function is working correctly, what is the result of the following call to `puzzleSearch`, using the above score function?

```haskell
  puzzleSearch ["*...", ".*..", "...*", ".*\."
```