# exercises
# Functors and I/O

**Exercise 1:**

Give a definition of the function

```haskell
fmap :: (a-> b) -> IO a -> IO b
```

the effect of which is to transform an interaction by applying the function to its result. You should define it using the

```haskell
do
```

construct.

**Solution 1:**

```haskell
fmap :: (a-> b) -> IO a -> IO b

fmap f m
  = do x <- m
       return (f x)
```

**Exercise 2:**

Define the function

```haskell
repeat :: IO Bool -> IO () -> IO ()
```

so that

```haskell
repeat test oper
```

has the effect of repeating ***oper*** until the condition ***test*** is ***True***.

**Solution 2:**

```haskell
repeat :: IO Bool -> IO () -> IO ()

repeat test m
  = do res <- test
       if res
          then return ()
          else do m
                  repeat test m
```

**Exercise 3:**

Define the higher-order function ***whileG*** in which the condition and the operation work over values of type ***a***. Its type should be:

```
whileG :: (a -> IO Bool) -> (a -> IO a) -> (a-> IO a)
```

so that

```
whileG cond  op x
```

has the effect of repeating ***op x*** while the condition ***cond x*** is ***True***.

**Solution 3:**

```
whileG :: (a -> IO Bool) -> (a -> IO a) -> (a -> IO a)

whileG cond op x
  = do test <- cond x
       if test
          then do op x
                  whileG cond op x
          else return x
```

**Exercise 4:**

Using the function ***whileG*** or otherwise, define an interaction which reads a number, ***n***, say, and then reads a further ***n*** numbers and finally returns their average.

**Solution 4:**

```
findAvg :: IO Integer

findAvg
  = do n <- getInt
       s <- sumInts n 0
       return (s `div` n)

sumInts :: Integer -> Integer -> IO Integer

sumInts n s
  = if n>0
       then do m <- getInt
               sumInts (n-1) (s+m)
       else return s
```

**Exercise 5:**

1. Define a function

```
accumulate :: [IO a] -> IO [a]
```

which performs a sequence of actions and accumulates their result in a list.
You can test this using (see testf below), e.g.:

```
> accumulate [readLn, testf "hi", readLn]
```

2. Also define a function :

```
sequence' :: [IO a] -> IO ()
```

which performs the interactions in turn, but discards their results.
You can test this using (see testf below), e.g.:

```
> sequence' [putStrLn "hello" , putStrLn "goodbye"]
```

3. Finally show how you would sequence a series, passing in values from one to the next :

```
seqList :: [a-> IO a] -> a -> IO a
```

***Hint:*** Use a simple function e.g

```
testf :: String -> IO String
testf x = do
            putStrLn x
            return (x ++ x)
```

(which takes a parameter and appends it to itself. It works on Strings).
So, you could call it as

```
> seqList [testf, testf, testf] "hello"   -- and get back
hello                                     --from first call (as IO effect)
hellohello                                --from second call
hellohellohellohello                      --from third call
"hellohellohellohellohellohellohellohello"   -- returned from function
```

**Solution 5:**

```
accumulate :: [IO a] -> IO [a]
accumulate [] = return []
accumulate (a:as) = do
                      x<- a
                      xs <- accumulate as
                      return (x:xs)
```

```haskell
--test this using
> accumulate [readLn, testf "hi", readLn]

sequence' :: [IO a] -> IO ()
sequence' [] = return ()
sequence' (a:as) = do
                    a
                    sequence' as
                    return()

 --test this with
 > sequence' [putStrLn "hello" , putStrLn "goodbye"]

seqList ::  [a-> IO a]  -> a -> IO a
seqList [] elem = return elem

seqList (a:as) elem = do
                       x <- a elem
                       seqList as x
```

**Exercise 6:**

Given the type definition

```haskell
data Result a = Succeed a | Fail
```

show how ***Result*** can be made into a monadic type.

**Solution 6:**

```haskell
  data Result a = Succeed a | Fail deriving (Eq, Show)

instance Functor Result where
  fmap f (Succeed x) = Succeed (f x)
  fmap _ _           = Fail

instance Applicative Result where
  pure = Succeed
  Fail <*> _ = Fail
  (Succeed f) <*> something = fmap f something

instance Monad Result where
  return = Succeed
  Succeed x >>= f = f x
  Fail >>= _ = Fail

-- to test this
```

```haskell
divBy :: Int -> Int -> Result Int
divBy 0 _ = Fail
divBy x y = Succeed (y `div` x)
```