IN3043 Functional Programming
Solutions to Exercises 10

1. As we know from an earlier exercise, a string is a palindrome if it is equal to its reverse, so we can write:

```
module Main where

main :: IO ()
main = do
    putStr "Enter a line: "
    line <- getLine
    if reverse line == line
        then putStrLn "palindrome"
        else putStrLn "not a palindrome"
```

   In the last part, we always do **putStrLn**, but its argument varies, so we could rewrite this part as

```
    putStrLn (if reverse line == line
                then "palindrome"
                else "not a palindrome")
```

2. Most of the work here is in a pure expression involving the contents. We need only wrap that using **readFile** and **putStr**:

```
printReversed :: FilePath -> IO ()
printReversed file = do
    contents <- readFile file
    putStr (unlines (reverse (lines contents)))
```

3. As a **do**-expression:

```
listCurrent :: IO ()
listCurrent = do
    files <- getDirectoryContents "."
    putStr (unlines files)
```

   or using the **>>=** operator directly:

```
listCurrent = getDirectoryContents "." >>= putStr . unline
```

4. First, a function to print one file:

```
printFile :: FilePath -> IO ()
printFile file = do
    contents <- readFile file
    putStr contents
```

   Alternatively, we could define it as

```
printFile = readFile file >>= putStr
```

   Now to apply this function to each of the files in current directory:

```
printAll :: IO ()
printAll = do
    files <- getDirectoryContents "."
    sequence_ [printFile file | file <- files]
```

   Alternatively, we could write **`printAll`** as

```
printAll = do
    files <- getDirectoryContents "."
mapM_ printFile files
```

   or just

```
printAll = getDirectoryContents "." >>= mapM_ printFile
```

5. We can write this from scratch:

```
repeatIO 0 action = return ()
repeatIO n action = do
    action
    repeatIO (n-1) action
```

   or equivalently

```
repeatIO 0 action = return ()
repeatIO n action = action >> repeatIO (n-1) action
```

   but we can also construct a sequence of **`n`** copies of the action, and use **`sequence_`** to execute them:

```
repeatIO n action = sequence_ (replicate n action)
```

6. In this case, if the user enters a non-palindrome, we need to do the same thing again. We can achieve this by defining a recursive action, which we call **`readLines`**.

```haskell
module Main where

main :: IO ()
main = do
    putStrLn "This program reads lines until a palindrome
       is entered"
    readLines

readLines :: IO ()
readLines = do
    putStr "Enter a line: "
    line <- getLine
    if reverse line == line
        then putStrLn "palindrome"
        else do
            putStrLn "not a palindrome"
            readLines
```