

PRELAB 4

1 Problem1

- (a) We can use `do` syntax to glue together several I/O actions into one. In a `do` block, the last action cannot be bound to a name
- (b) Open up terminal and navigate to the directory where our file is located. Then call the command: — `make < executablename >`. We run the executable file by `./executablename`.
- (c) `name<- getLine` means Perform the I/O action `getLine` and then bind its result value to `name`. `name = getLine` reads a line from `getLine` and stores it into a variable called `name`. There is a difference as `getLine` operation would return some string type, while `=` operator assumes that `getLine` is defined. To get the value out of an I/O action, you have to perform it inside another I/O action by binding it to a name with `<-`.
- (d) I understand that `return` wraps up a pure type (a specific type) into a (dummy) IO operation so that Haskell can perform it when outputting something would cause errors. Using `return` doesn't cause the I/O `do` block to end in execution or anything like that.
- (e) `map` takes a function and a list and applies that function to every element in the list, producing a new list. Because mapping a function that returns an I/O action over a list and then sequencing it is so common, the utility functions `mapM` and `mapM_` were introduced. `mapM` takes a function and a list, maps the function over the list and then sequences it. `mapM_` does the same, only it throws away the result later. We usually use `mapM_` when we don't care what result our sequenced I/O actions have.
- (f) `main = putStrLn "Hello world!!"`
- (g) `openFile :: FilePath -> IOMode -> IO Handle`.
- (h) `ReadMode | WriteMode | AppendMode | ReadWriteMode : 4`
- (i) `WithFile` takes a path to a file, an `IOMode` and then it takes a function that takes a handle and returns some I/O action. What it returns is an I/O action that will open that file, do something we want with the file and then close it. The result encapsulated in the final I/O action that's returned is the same as the result of the I/O action that the function we give it returns. In other words - `withFile` opens the file and then passes the handle to the function we gave it. It gets an I/O action back from that function and then makes an I/O action that's just like it, only it closes the file afterwards.
`openFile` takes a file path and an `IOMode` and returns an I/O action that will open a file and have the file's associated handle encapsulated as its result. It does not close the file so we have to do it manually.
`readFile` takes a path to a file and returns an I/O action that will read that file (lazily, of course) and

bind its contents to something as a string. It's usually more handy than doing `openFile` and binding it to a handle and then doing `hGetContents`. We cannot close the file manually, therefore Haskell does that for us.

(j) `import Data.Char`

```
main = do
  contents <- getContents
  putStr(map toUpper contents)
```

(k) `main = do`

```
  line <- getContents
  putStr( countSpaces line)
```

```
  countSpaces [] = 0
  countSpaces [x:xs] = |if x==" " | x=="\n" | x=="\r" = 1+countSpaces xs
  otherwise
  0+countSpaces xs
```

(l) `import System.IO import Data.Char`

```
main = do contents <- readFile "girlfriend.txt"
  writeFile "girlfriendcaps.txt" (map toUpper contents)
```