HASKELL I/O

Curt Clifton
Rose-Hulman Institute of Technology

Please SVN Update your *HaskellInClass* folder, then open *eieio.hs*

SEPARATION OF CONCERNS

- Haskell separates pure code from side-effecting code
  - Helps us reason about programs
  - Allows compiler to aggressively optimize/parallelize pure code

EXAMPLE I/O IN HASKELL

keyword, introduces a **sequence** of actions

assignment, unpacks result of getLine action

```
ex1 = do
    putStr "WHAT is your name? "
    inpStr1 <- getLine
    putStr "WHAT is your quest? "
    inpStr2 <- getLine
    putStrLn ("Good luck with that, " ++ inpStr1 ++ "!")
```

```
ghci> :type putStr
putStr :: String -> IO ()
ghci> :type getLine
getLine :: IO String
ghci> :type ex1
ex1 :: IO ()
```

Anything of type IO *something* is an IO ***action***

Q1

CALLING PURE CODE FROM ACTIONS

unpacks results from actions

```
transform :: String -> String
transform s = s ++ " is a lovely shade for a coconut."

ex2 :: IO ()
ex2 = do
    putStr "WHAT is your favorite color? "
    inpStr <- getLine
    let outStr = transform inpStr
    putStrLn outStr
```

within *do*, use *let* (without *in*) to get results from pure code

PURE	IMPURE
Referentially transparent	Different results for same parameters are possible
No side effects	May have side effects
Never alters state	May alter global state of the program, system, or world

Q2

FILE I/O

```
fileTransform :: IO ()
fileTransform = do
    inHandle <- openFile "eieio.hs" ReadMode
    outHandle <- openFile "shout.txt" WriteMode
    mainLoop inHandle outHandle
    hClose inHandle
    hClose outHandle

mainLoop :: Handle -> Handle -> IO ()
mainLoop inh outh = do
    atEOF <- hIsEOF inh
    if atEOF
        then return ()
        else do line <- hGetLine inh
                hPutStrLn outh (map toUpper line)
                mainLoop inh outh
```

return wraps a pure value in IO, opposite of <-

Q3–5

LAZY I/O

- *hGetContents :: Handle -> IO String*
  - "Reads" entire file into String **lazily**
  - Like Python's *read*, but no memory leak…
  - …as long as we just use result once

SIMPLER STILL

- ghci> :type readFile
  readFile :: FilePath -> IO String
  ghci> :type writeFile
  writeFile :: FilePath -> String -> IO ()

- bestFileTransform :: IO ()
  bestFileTransform = do
      inContents = readFile "eieie.hs"
      writeFile "shout.txt" (map toUpper inContents)

MISCELLANEOUS I/O HELPERS

- *interact :: (String -> String) -> IO ()*
  - Reads from *stdio*, applies argument function, writes to *stdout*
- *hTell*, *hSeek*: find/set position in file
- Predefined handles: *stdin*, *stdout*, *stderr*
- *System.Directory* module:
  - *removeFile*, *renameFile*, *getTemporaryDirectory*
- *openTempFile*
- *System.Environment* module:
  - *getArgs*, *getProgName*, *getEnv*

EXERCISE

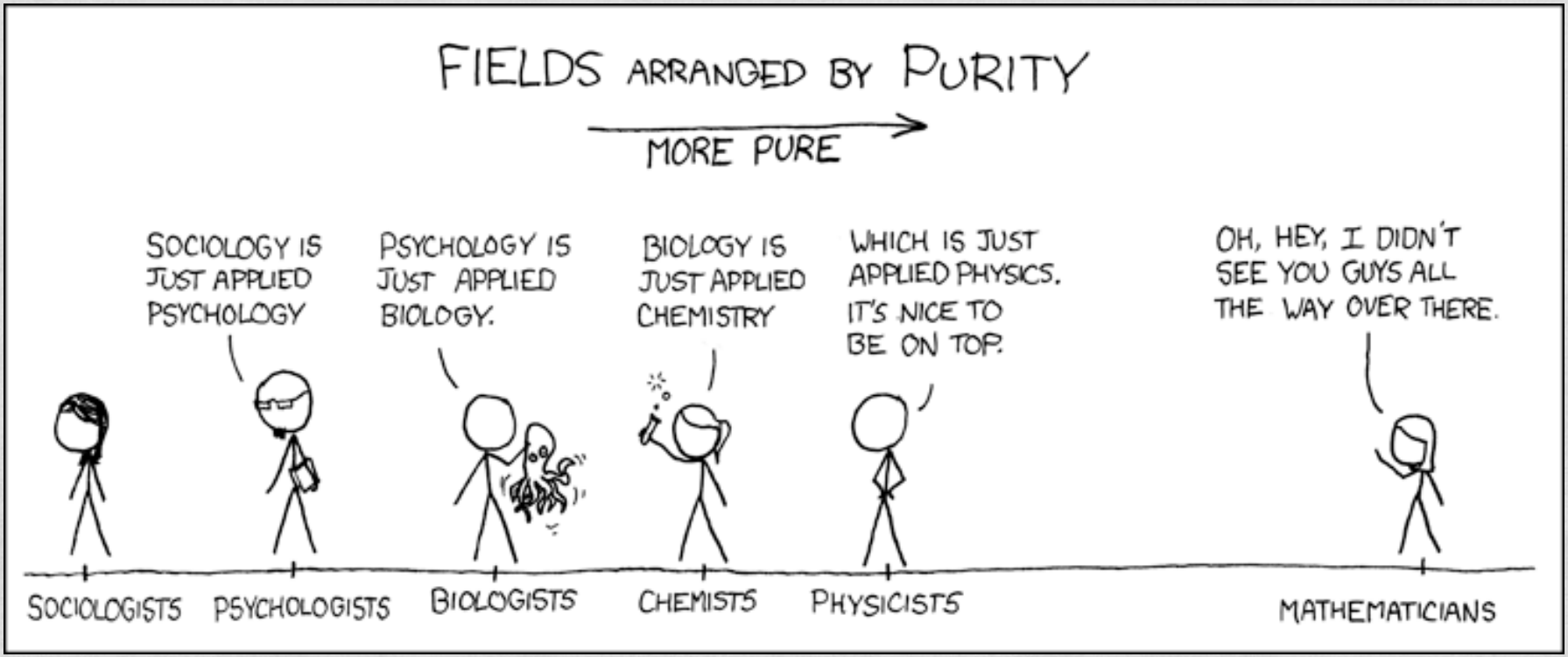
Implement an I/O action, *wordProcessor :: IO ()*, that prompts the user for a series of words and prints a count of the words entered, along with the longest and shortest words. For example:

```
ghci> wordProcessor
Enter a word, or just return to quit: dog
Enter a word, or just return to quit: cat
Enter a word, or just return to quit: whale
Enter a word, or just return to quit: raptor
Enter a word, or just return to quit:
Number of words: 4
Longest word: raptor
Shortest word: cat
```

The pure helper functions *longest* and *shortest* are provided.

THE IO MONAD

PURITY



You'll have to look up the alt text ;-)

CAN WE BE JUST A LITTLE BIT IMPURE?

- How are we getting side effects if Haskell is a pure language?
- Solution: Pass along an object to be "mutated"
- Original: *f* :: *Tree* → *Int*
- New: *f* :: *(Tree, State)* → *(Int, State)*

Monads automate this pattern

Original State

"Mutated" State

MONADIC MAPS

```
strToMessage :: String -> String
strToMessage s = "… sir: " ++ s

putMessage :: String -> IO ()
putMessage = putStrLn . strToMessage

strings = ["Lancelot", "Robin"]

ex3 = do
    putMessage "Start me up"
    mapM_ putMessage strings
    putMessage "That's all folks!"
```

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

THE MONAD TYPECLASS

Sequences two expressions that have Monad results

Sequences two Monad expressions binding result of first for use in second

```
class Monad m where
  (>>) :: m a -> m b -> m b
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail :: String -> m a
```

Wrap pure value in Monad

Q6

DA DO DO DO

- The *do* expression in Haskell is just a sugar for Monad sequencing

Inside *do*	Monad notation
`e1` `e2`	`e1 >>= \_ -> e2` or `e1 >> e2`
`x <- e1` `e2`	`e1 >>= \x -> e2`
`return e1`	`return e1`

SUGAR FREE!

```
ex4 = do
    putStr "WHAT is your name? "
    inpStr1 <- getLine
    putStrLn ("Bugger off, " ++ inpStr1 ++ "!")
```

desugar

```
ex5 =
    putStr "What is your name? " >>
    getLine >>=
        (\inpStr -> putStrLn ("Bugger off, " ++ inpStr ++ "!"))
```

desugar

```
ex6 =
    putStr "What is your name? " >>=
        (\_ -> getLine >>=
                (\inpStr -> putStrLn ("Bugger off, " ++ inpStr ++ "!")))
```