

HASKELL I/O

Curt Clifton

Rose-Hulman Institute of Technology

SVN update then look in *HaskellIO* folder
and 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
  putStrLn "WHAT is your name? "
  inpStr1 <- getLine
  putStrLn "WHAT is your quest? "
  inpStr2 <- getLine
  putStrLnLn ("Good luck with that, " ++ inpStr1 ++ "!")
```

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

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

QI

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

Referentially transparent

No side effects

Never alters state

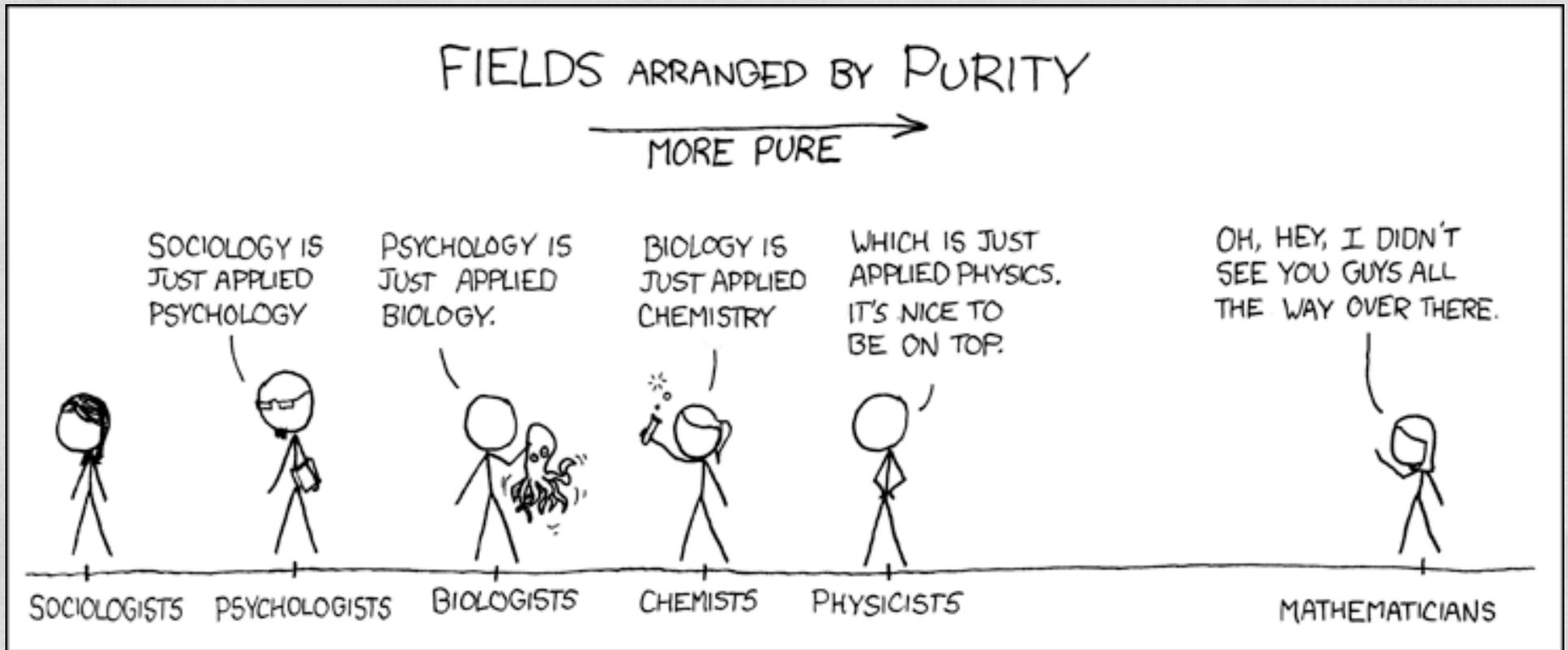
IMPURE

Different results for same parameters are possible

May have side effects

May alter global state of the program, system, or world

PURITY



<http://xkcd.com/435/>

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

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 <-

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.