# Programming In Haskell Chapter 8

CS 1JC3

# Pure Functions

- Pure Functions have two important properties we should make note of
  - Have no Side Effects (functions always return the same result on the same input and variables are immutable)
  - Can be Lazy (nothing gets evaluated until it has to be)

# Pure Functions

- Pure Functions have two important properties we should make note of
  - Have no Side Effects (functions always return the same result on the same input and variables are immutable)
  - Can be Lazy (nothing gets evaluated until it has to be)

- Impure Functions allow side effects, take for example the following impure C code

```c
int counter = 0;
int return_global_counter(int a)
{
  return counter++;
}
```

# Pure Functions

To illustrate important properties of Haskell functions, consider the following code

```
uselessArithmetic x y = let
   -- order of square1,square2,square3 doesn't matter
   square3 = square2 * x
   square1 = x
   square2 = square1 * x
 in square3

uselessArithmetic2 = let
   x = 1
   y = sum [1..] -- never gets evaluated
 in uselessArithmetic x y
```

# Your First IO Function: print

The following function prints the line Hello World! to the standard output

```
print "Hello World!"
```

Note: This is different than ghci just regurgitating values of whatever expression it last evaluated, it explicitly prints its argument and is needed to output anything for compiled programs

# Your First IO Function: print

The following function prints the line Hello World! to the standard output

```
print "Hello World!"
```

Note: This is different than ghci just regurgitating values of whatever expression it last evaluated, it explicitly prints its argument and is needed to output anything for compiled programs

Question: Is print a pure function? Why / why not?

# Your First IO Function: print

What's the type of print?

# Your First IO Function: print

What's the type of print?

```
print :: Show a => a -> IO ()
```

- Think of IO as a special data constructor, one that you never try to pull a value out of directly
- Think of () as the empty type
- print doesn't just return nothing, it returns an IO value so any function that calls it must be able to process IO values

# Sequencing IO Functions

- Haskell functions inherently are evaluated like expressions and lazily, without any explicit sequencing of computation.
- But with IO we need some way of sequencing, i.e saying output this, then output this, etc

# Sequencing IO Functions

- Haskell functions inherently are evaluated like expressions and lazily, without any explicit sequencing of computation.

- But with IO we need some way of sequencing, i.e saying output this, then output this, etc

- Enter the do syntax

```
printNonsense :: IO ()
printNonsense = do { print "Output this";
                     print "then output this";
                     print "etc" }
```

Note: the curly braces and semi-colin's are not necessary if you follow the alignment rule

# More IO Functions: Output

```
-- prints a String (without a newline)
putStr :: String -> IO ()

-- prints any type with a Show instance
print :: Show a => a -> IO ()

-- writes a string to a file (creates / overwrites file)
writeFile :: FilePath -> String -> IO ()

-- appends a string to a file (file must already exist)
appendFile :: FilePath -> String -> IO ()
```

Note: The type FilePath is *basically* a String type synonym

# More IO Functions: Input

```haskell
-- gets a line of input and returns it as a String
getLine :: IO String

-- get a single character from input
getChar :: IO Char

-- reads a file’s contents and returns it as a String
readFile :: FilePath -> IO String
```

# Using IO Input Functions

```
echo :: IO ()
echo = let
        line = getLine
    in print line
```

WRONG! Why does this cause an error? Whats the type of line?

# Using IO Input Functions

```
echo :: IO ()
echo = let
        line = getLine
    in print line
```

WRONG! Why does this cause an error? Whats the type of line?

```
echo :: IO ()
echo = do { line <- getLine;
            print line }
```

CORRECT! Whats the type of line here?

# The Subtleties of IO and do Notation

- Every line of a do structure must return some IO a

# The Subtleties of IO and do Notation

- Every line of a do structure must return some IO a

- The ← operator takes a function of type IO a and extracts the a value

# The Subtleties of IO and do Notation

- Every line of a do structure must return some IO a

- The ← operator takes a function of type IO a and extracts the a value

- The last line defines the final return type and cannot be a use of the ← operator

# The Subtleties of IO and do Notation

- Every line of a do structure must return some IO a
- The ← operator takes a function of type IO a and extracts the a value
- The last line defines the final return type and cannot be a use of the ← operator
- Any function that calls an IO function must also be an IO function

# The return Function

The return function is used for wrapping a value as an IO

```
return :: a -> IO a
```

# The return Function

The return function is used for wrapping a value as an IO

```
return :: a -> IO a
```

You can use it for returning pure values in an IO function

```
get2Lines :: IO String
get2Lines = { ln1 <- getLine;
              ln2 <- getLine;
              return (ln1 ++ ln2) }
```

Disclaimer: If you are a real Haskell coder reading this, please forgive me for pretending return is esoteric to IO

# IO and Recursion

We can create a program that repeats an IO action (in this case forever) using recursion

```
echoForever :: IO ()
echoForever = do { line <- getLine;
                   print line;
                   echoForever }
```

Question: How do we make this function stop?

# IO and Recursion

We can create a program that repeats an IO action (in this case forever) using recursion

```
echoForever :: IO ()
echoForever = do { line <- getLine;
                   print line;
                   echoForever }
```

Question: How do we make this function stop?
Hint: Use return ()

# The show Function

The show function takes any type thats an instance of the Show class and converts it to a String

```
show :: Show a => a -> String
```

# The show Function

The show function takes any type thats an instance of the Show class and converts it to a String

```
show :: Show a => a -> String
```

Useful for outputting results with print

```
add :: Num a => a -> a -> a
add x y = x + y

main :: IO ()
main = do print ("5 + 4 = " ++ show (add 5 4))
```

# The read Function

The read function is basically the inverse of the show function, it takes a String and attempts to convert it to a different type

```
read :: Read a => String -> a
```

# The read Function

The read function is basically the inverse of the show function, it takes a String and attempts to convert it to a different type

```
read :: Read a => String -> a
```

It’s often best to specify the type being read explicitly, for example

```
addInts :: IO Int
addInts = do {x <- getLine;
              y <- getLine;
              return (addStrings x y)}
addStrings :: String -> String -> Int
addStrings x y = let
                   x’ = read x :: Int
                   xs = map read [x,y] :: [Int]
        in sum xs + 0*x’
```

# The lines / unlines Functions

- readFile returns a single String with newline characters to specify line separations
- The lines function

  ```
  lines :: String -> [String]
  ```

  takes a String and returns a list of Strings for each line
- The unlines function

  ```
  unlines :: [String] -> String
  ```

  is quite simply the inverse of lines

# PSA: Do Not Abuse IO

- The IO type is for doing IO related actions
- Make sure to separate out computation that can be done purely from IO

# PSA: Do Not Abuse IO

- The IO type is for doing IO related actions
- Make sure to separate out computation that can be done purely from IO

- Example of BAD CODE

```
bad_code = do { x  <- getLine;
                x’ <- return (read x :: Int);
                y  <- getLine;
                y’ <- return (read y :: Int);
                z  <- return (x’ + y’);
                z’ <- return z;
                print z’ }
```

# PSA: Do Not Abuse IO

For reference, BETTER CODE

```
addStrings :: String -> String -> String
addStrings x y = let
       x’ = read x :: Int
       y’ = read y :: Int
    in x’ + y’

better_code :: IO ()
better_code = do { x <- getLine
                   y <- getLine
                   print (addStrings x y) }
```

# The main Function

- Compiled Programs need a main function that will be the root function of your program

```
main :: IO ()
```

# The main Function

- Compiled Programs need a main function that will be the root function of your program

```
main :: IO ()
```

- main is always of type IO ()

# The main Function

- Compiled Programs need a main function that will be the root function of your program

```
main :: IO ()
```

- main is always of type IO ()
- Usually put in a root module called Main in a file Main.hs

# The main Function

- Compiled Programs need a main function that will be the root function of your program

```
main :: IO ()
```

- main is always of type IO ()
- Usually put in a root module called Main in a file Main.hs
- Since we will only be using ghci in this course, we will never need to make a main function, but it's good to know of it's existence

# Exercise 1

Redefine the echoForever function so that it stops when the user enters quit. Call the function echoTillQuit

Recall:

```
echoForever :: IO ()
echoForever = do { line <- getLine;
                   print line;
                   echoForever }
```

# Solution 1

```
echoTillQuit :: IO ()
echoTillQuit = do { line <- getLine;
                    print line;
                    if line == "quit"
                        then return ()
                        else echoTillQuit }
```

# Exercise 2

Write some code that

- has a root IO function main
- creates a file log.txt
- gets user input and appends it to the file log.txt
- repeats until the user enters quit

# Solution 2

```
main :: IO ()
main = do writeFile "log.txt" ""
          getLineAndLog
```

# Solution 2

```haskell
main :: IO ()
main = do writeFile "log.txt" ""
          getLineAndLog

getLineAndLog :: IO ()
getLineAndLog = do inp <- getLine
                   if inp == "quit"
                      then return ()
                      else logAndLoop inp
```

# Solution 2

```haskell
main :: IO ()
main = do writeFile "log.txt" ""
          getLineAndLog

getLineAndLog :: IO ()
getLineAndLog = do inp <- getLine
                   if inp == "quit"
                        then return ()
                        else logAndLoop inp

-- getLineAndLog and logAndLoop are "mutually-recursive"
logAndLoop :: IO ()
logAndLoop out = do appendFile "log.txt" out
                    getLineAndLog
```

# Exercise 3

Write some code that

- Reads a file Ints1.txt that you assume contains an Int on each line
- Converts the file contents to a [String] (Hint: use lines)
- Reads the list as [Int] (i.e map the read function)
- Sorts the Ints (use one of the sorting functions you defined last tutorial)
- Write your newly sorted Ints to a new file Ints2.txt, line by line

Note you need to create a file Ints1.txt with one Int each line in the same directory as your code

# Solution 3

```haskell
-- IO Code
main :: IO ()
main = do inp <- readFile "Ints1.txt"
          writeFile "Ints2.txt" (parseAndSort inp)
```

# Solution 3

```haskell
-- IO Code
main :: IO ()
main = do inp <- readFile "Ints1.txt"
          writeFile "Ints2.txt" (parseAndSort inp)

-- Pure Code
parseAndSort :: String -> String
parseAndSort inp = let
      strings  = lines inp
      ints     = map read strings :: [Int]
      sorted   = mergeSort ints
      strings' = map show sorted
  in unlines strings'

mergeSort :: (Ord a) => [a] -> [a]
 ...
```

# Exercise 4

Write some code that

- starts at a sum zero
- gets a line of user input (prompt the user to do so with putStr)
- reads the line assuming it's an Int
- adds the last input to the current sum
- prints the sum and REPEATS

# Solution 4

```haskell
main :: IO
main = getIntAndSum 0

getIntAndSum :: Int -> IO ()
getIntAndSum x = do putStr "Input Integer: "
                    int <- getLine
                    printAndLoop (x + read int)

printAndLoop :: Int -> IO ()
printAndLoop x = do putStr "Current Sum: "
                    print x
                    getIntAndSum x
```

# Exercise 5

Write some code that

- create a variable questions :: [(String,String)] that contains a list of Yes/No questions and their solutions as Strings
- write code that iterates through the list, asks the user each question and tells them if they got the right answer

Example question list:

```haskell
questions = [("Does 1+2=3: ","yes")
            ,("Does 5/0=5: ","no")]
```

# Solution 5

```
questions = [("Does 1+2=3: ","yes")
            ,("Does 5/0=5: ","no")]

main :: IO ()
main = playQuestions questions

playQuestions :: [(String,String)] -> IO ()
playQuestions []         = print "Game Over"
playQuestions ((q,a):qs) = do putStr q
                              a’ <- getLine
                              if a == a’
                                then print "Correct!"
                                else print "Wrong!"
                              playQuestions qs
```