

## Week07 Presentation

Agenda:

1. BRIEFLY go through tutorial video content (VS CODE)
2. Hints for Exercises 05

### **Pure Functions:**

#### 1. **No Side Effects**

A side effect is produced when a program or function modifies the state of something else outside its own scope.  
i.e. modifying global variables

Essentially a side effect is produced by a function that depends or interacts with an external source

- a. Functions given the same input will always give the same output
- b. Functions in Haskell fully determined by its arguments

Example of side effects :

```
int glob = 0;
int square(int x)
{ glob = 1;
  return x*x;
}
int main()
{
    int res;
    glob = 0;
    res = square(5);
    res += glob;
    return res;
}
```

Here - global variable 'glob' is being changed within the 'square' function, it does something outside the scope of the original function.

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

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

Here - no matter what input we give this function, it'll return a different output.

No side effects:

F x = x  
F 5 =5

F 6 = 6

2. **Can be lazy**
  - a. Nothing gets evaluated until needed
  - b. Haskell won't evaluate arguments to a function before calling it. Unless proven otherwise, Haskell assumes that the arguments will not be used by the function, so it procrastinates as long as possible.

Add3 = undefined + 1

<-- haskell compiler won't complain that we are trying to add undefined to 1. The error that arises is a result from trying to evaluate undefined/

Foo x = 1

addFunc = foo(undefined) + 1

Haskell calls foo but never evaluates the its argument undefined. It realizes that the argument x is discarded from the function output.

**Arguments are not evaluated before they are passed to a function, but only when their values are actually used.**

```
def someFunction(x, y) =  
    return x + 1  
  
someFunction(3, 3 + 2)
```

i.e. in Haskell the expression '3+2' would not get evaluated because it's not used.

## 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 x
```

Things are evaluated in the order in which they are needed. It is not evaluated in the order which the programmer specifies them.

In other languages, the code would run, and it would reach y and try and evaluate this infinite list.

- GHCi automatically calls print for you

## Haskell: print Vs putStrLn

Haskell provides `putStr`, `putStrLn` functions, which write given string to standard output (terminal).

`putStr`, `putStrLn` are almost same, only difference is `putStrLn` adds a newline character.

```
Prelude> :t putStr
putStr :: String -> IO ()
```

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

`print` function displays value of any printable type to the standard output device.

Technically `print` is defined like below.

`Print` also adds a new line.

```
print x = putStrLn (show x)
```

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

Finally, If you have a `String` that you want to print to the screen, you should use `putStrLn`. If you have something other than a `String` that you want to print, you should use `print`.

\*\* All of the default types are in the `show` class

\*\* Any type you create, you can put 'deriving show' and it'll be apart of the `show` class.

```
-- 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 ()
```

## IO

- Special Data constructor - one that you never try to pull a value out of directly
- Print returns and IO value  
() empty type
- IO involves sequencing however haskell is LAZY
  - Therefore we use the 'do' syntax
- Any function that calls an IO function must also be an IO function

```
echoFile = do fileContents <- readFile "file.txt"
print fileContents
-- function echoFile calls print which is an IO function, therefore echoFile must also be a
IO function
```

## Do SYNTAX

- Specifies expressions in ORDER
- Every line of a do structure must return some IO a

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

\*\* don't need the curly braces, or semicolons as long as they line up.

Data typeName = Val1 | Val2 Int

## MORE IO FUNCTIONS: INPUT

```
-- 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
```

## (<-) Operator (Single assignment Operator)

- Takes a function of type IO a and extracts the a value
- Last line cannot use this operator

Remember: Haskell is a PURE language, meaning NO SIDE EFFECT

$F\ x = x$

$F\ 5 = 5$

$F\ 6 = 6$

However these IO Functions handle external sources, i.e.

- `readFile`, can call it with the same input (same file) and the output will change depending on the time and contents of the file.

**IO functions call Pure functions**

**Pure functions NEVER call IO functions -- if you did this the "pure" functions would not be pure anymore. Would technically have side effects**

`Func1 :: IO a -> a <----- DOES NOT WORK`

**Return** :: `a -> IO a <----` does work!

## THE RETURN FUNCTION

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

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

- You can use it for converting pure values in an IO

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

Show:

## THE SHOW FUNCTION

- The show function converts a value 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))
```

Return :

## THE READ FUNCTION

- The read function converts a String to a value

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

- Generally you need to explicitly specify the type

```
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'
```

Lines /Unlines

## THE LINES / UNLINES FUNCTIONS

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

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

- The unlines function is quite simply the inverse of lines

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

## Merge Sort

