Week07 Presentation

Agenda:

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

Pure Functions:

1. No Side Effects

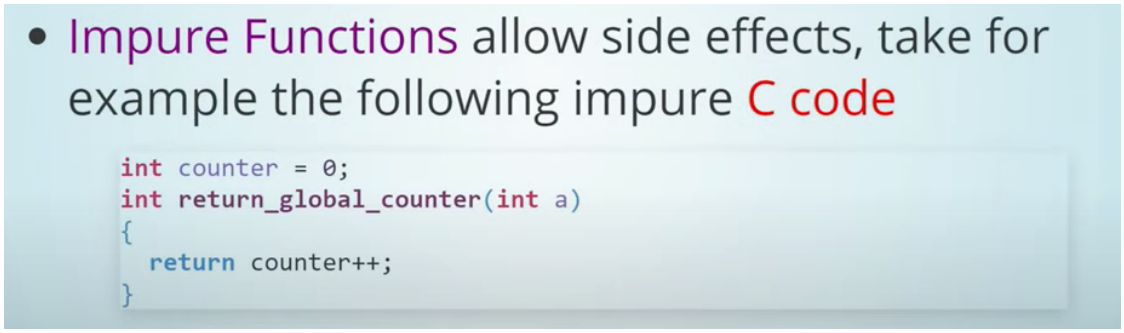
A side effect is produced when a program or function modifies the state of something else outside its own scope.
i.e. modifying global variables
Essentially a side effect is produced by a function that depends or interacts with an external source

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.



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

No side effects:

F x = x
F 5 =5

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

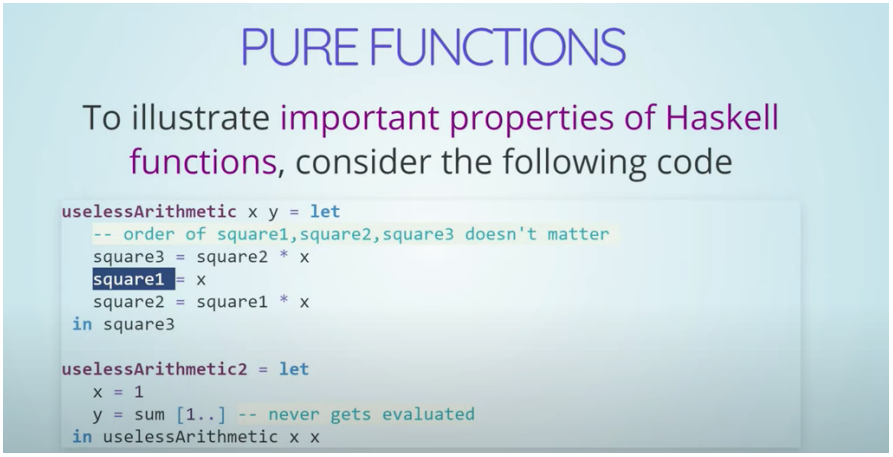
Haskell calls foo but never evaluates the its argument undefined. It realizes that the argument x is discarded from the function output.

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.



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 ()
```

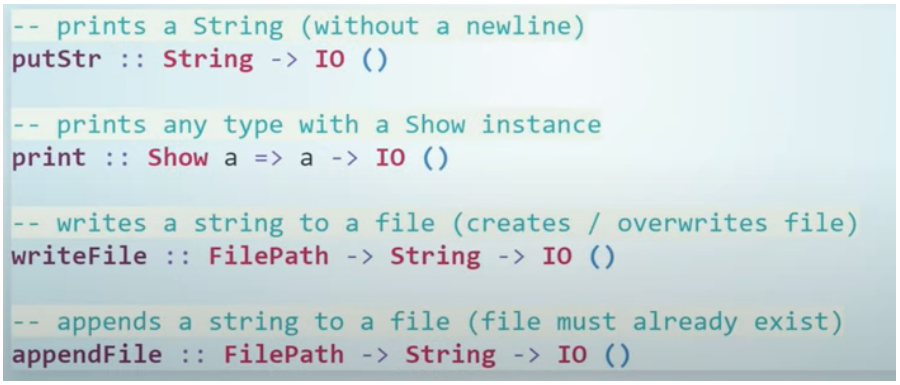
print function displays value of any printable type to the standard output device.
Technically print is defined like below.
Print also adds a new line.

```
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.



```haskell
-- 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
```

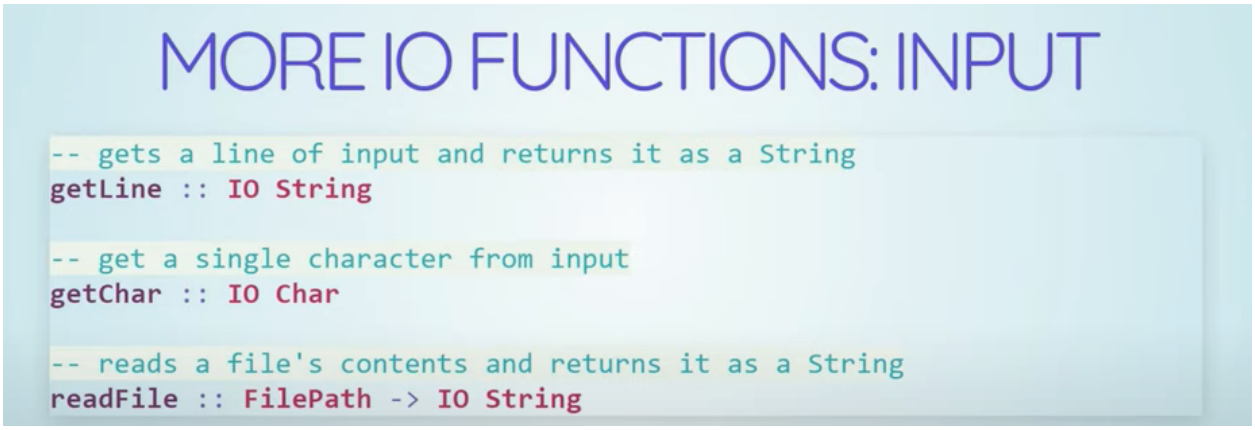
Do SYNTAX

- 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



(<-) 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

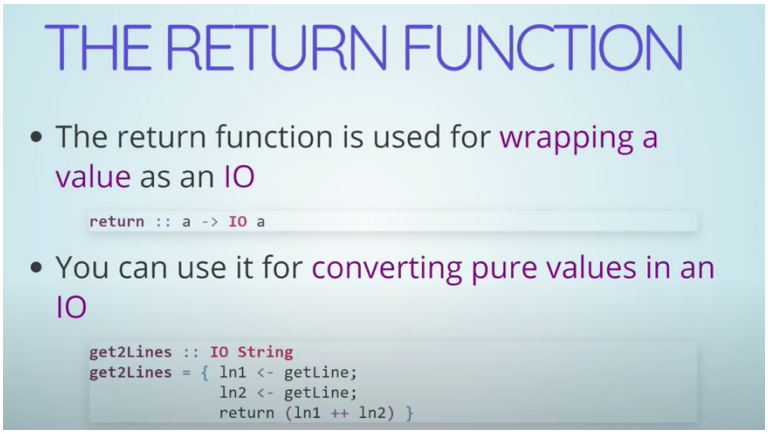
However these IO Functions handle external sources, i.e.

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

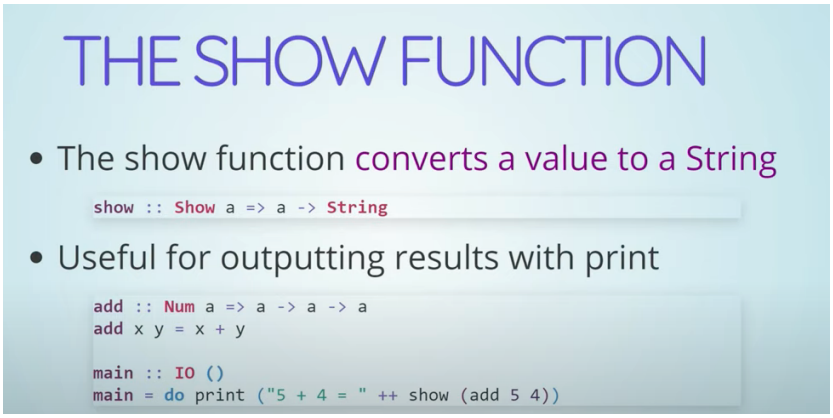
IO functions call Pure functions
Pure functions NEVER call IO functions -- if you did this the "pure" functions would not be pure anymore. Would technically have side effects

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

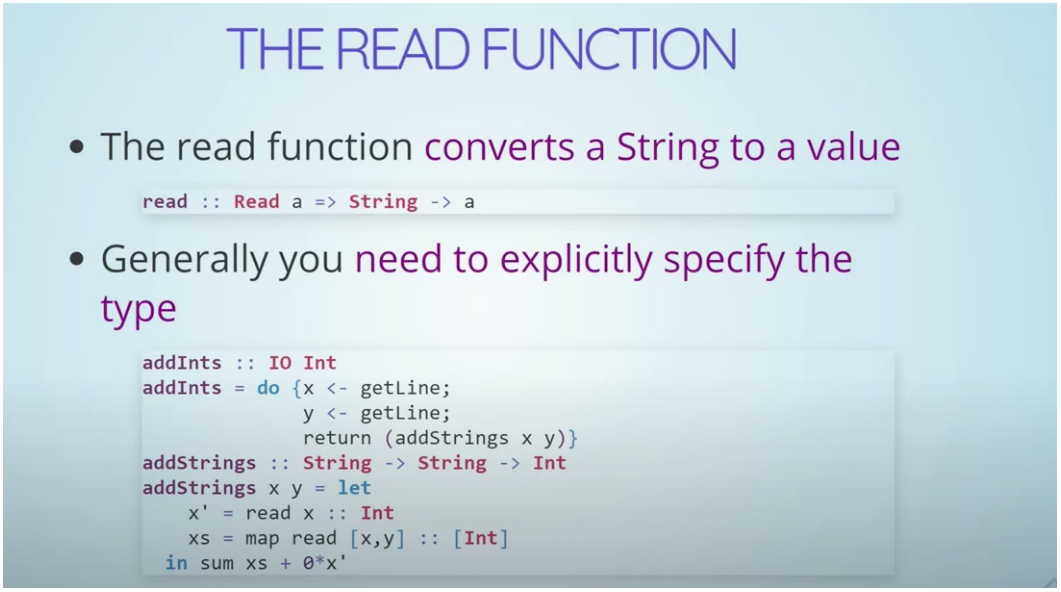
Return :: a -> IO a <<--- does work!



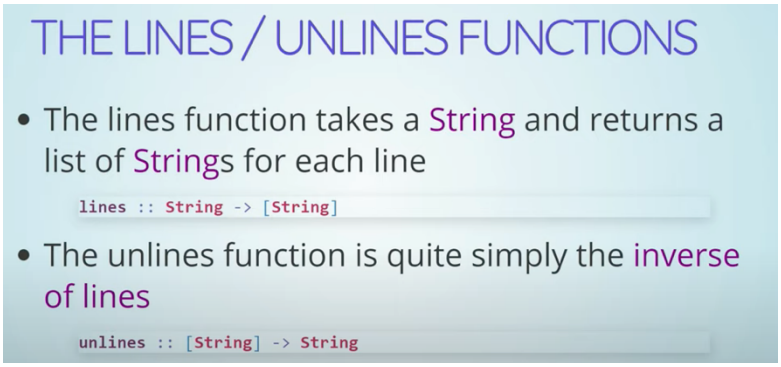
Show:



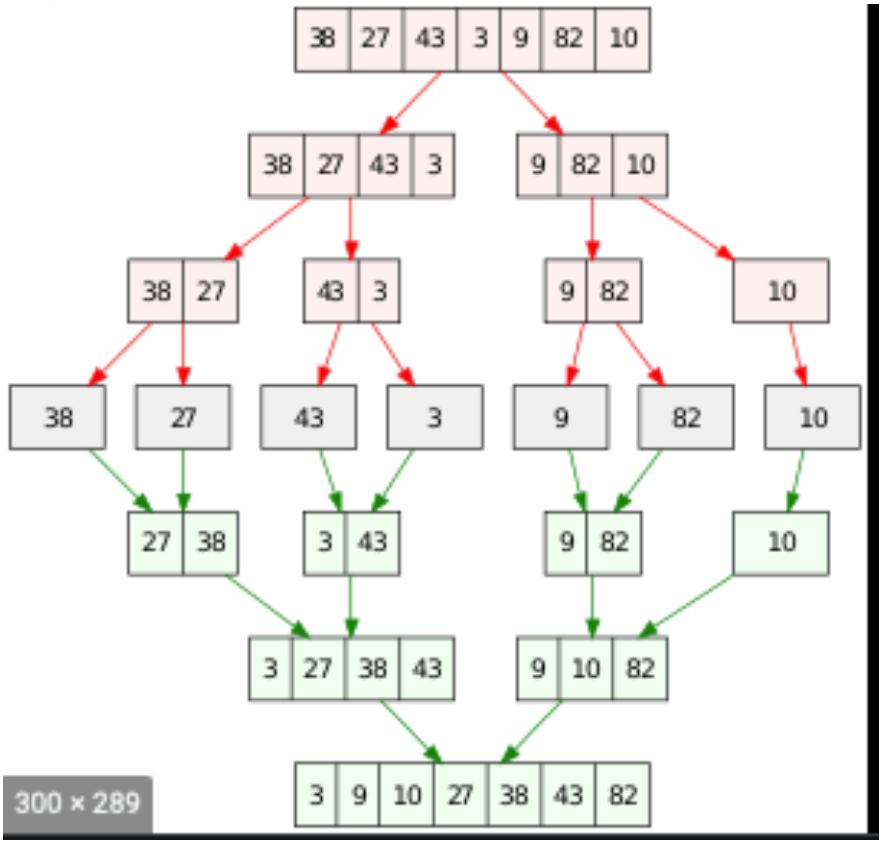
Return :



Lines /Unlines



Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82
300 × 289