# Lecture Four

> “To imagine a language is to imagine a form of life.”
> — Ludwig Wittgenstein

## 1. Introduction

*Currently Being Reviewed... Plan On Adding More Notes In The Future*

*Note: Combinational Lecture (4) and Parts Of LearnYouAHaskellForGreaterGood*

During this lecture, we're going to cover *a lot*. Firstly, we'll be briefly discussing the possibility and extent of on-chain arbitrary logic (even though this lecture focuses on a Haskell Primer and off-chain logic). So, we'll have a quick chat about: validation logic, plutus script, network nodes, context, IO and native tokens. When it comes to the meat and potatoes, the off-chain code and the Haskell primer... We're going to examine what monads are, specifically the contract monad, as implemented by the wallet. Furthermore, we'll review some off-chain checks / validation, then building the transactions themselves too (to be deployed to our own / nearest node). Oh yes, how did I forget. We'll also be discussing safe vs unsafe and useful vs non-useful. Lastly, IO (that stands for Input Output, just in case you didn't know) and the effects that can be had using Haskell IO.

## 2. An 'Advanced' Haskell Primer

As I have previously stated, I have done **some** Haskell during my undergraduate degree and then a refresher during my postgrad. However, it has been quite a while; and if I'm completely honest, I never wrote **that much**. I mean, when you compare the couple of hundred (potentially, I can't remember) lines of Haskell I wrote with the tens of thousands of lines of ARM assembly I was forced to write, well, it puts things in perspective. If anyone needs an ARM assembly guy, I'm your man.

**Some Important Points To Consider**

- According to Lars, Monads are very similar to burritos; and guess what, I **LOVE** burritos.
- Monads are usually the first stumbling block for people who are not use to Haskell.
- We're going to enter into Haskell **very gently within these notes**.
- Then, we'll provide a brief introduction to Monads.
- **HOWEVER...** Before we get to Haskell, let's look at (the dreaded) Java (Unsafe and

Useful!)

## 2.1 Imagine: Mainstream A "Unsafe" "Useful" Language: Java

Firstly, when it comes to imperative programming[1] a single function may do one of many things[2]. In the world of 'Smart Contracts' which may be dealing with peoples' livelihoods and quite importantly, their money, it's a good idea to try and limit the amount of 'things' such a contract is able to do. Essentially, go and watch the Simon Peyton Jones video about safe and unsafe programming languages [2], it is somewhat tongue in cheek (at least, the title was at the time), but there is **a lot** of sense to it. Please see below for an example of an "unsafe" "useful" programming language.

**Java Example:**

```
import java.lang.Integer;

Public Static I makeSomethingHappen() {
    // ...
}
...
System.out.println(makeSomethingHappen();
...
makeSomethingHappen();
```

Even though the above code will always return a value of type int, since we do not know what the ... executes (if it was not a comment, of course!), it may not always return the same `int` value. This is due to the fact that the Java program could be performing arbitrary input/output, it may depend on any number of, say: global variables, connections to a DB, API calls, etc. Essentially: it's unpredictable in nature. It's unsafe. But, it can be pretty useful (Thanks Simon!).

### 2.1.1 A More Concrete Unsafe (But Rather Non-Useful) Example

To demonstrate what we're talking about here more concretely, I wiped out my old Java book from about ten years ago and threw together the following code. Now, prepare to be disgusted, because Java programmers use notoriously long variable names. You could even say: the polar opposite of Haskell programmers (as most of them are mathematicians, so `x` seems to be perfectly adequate to provide a useful naming convention for mathematicians)! Anyway, see below *(and yes, Java is unnecessarily verbose)*.

```
import java.net.URL;
import java.net.HttpURLConnection;
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
import java.lang.StringBuilder;
import java.net.MalformedURLException;
import java.io.IOException;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class Unsafe
{
    // Yes - code should be refactored, it's an example.
    /**
     * Get the HTTP repsonse body (in string format) at a spec'd URL
     *
     * Usually this would be refactored, but this is a simple example
     * In addition, there are ample comments here, so.. do one.
     *
     * @param fromUrl A string: HTTP endpoint
     * @return        The contents at the fromUrl param
     */
    public static String getSingleLineHttpResponse(String fromUrl)
    {
        // typical Java; x aint good enough we have to use long var names
        // the code should read such that it doesn't need comments
        // ref: Uncle Bob: Clean Code
        try
        {
            URL webAddress = new URL(fromUrl);
            HttpURLConnection webConnection = (HttpURLConnection) webAddr
            if(webConnection.getResponseCode() != 200)
            {
                throw new IOException("Error: Sorry, but the connection w
            }
            BufferedReader someMemReader = new BufferedReader(new InputSt
            return someMemReader.readLine();
        }
        catch(MalformedURLException malformedException)
        {
            return "Error: the supplied URL is malformed or.. BAD";
        }
        catch(IOException inputOutputException)
        {
            return "Error: the input resource is unavailable or.. BAD";
        }
    }

    /**
     * Simply removes an expected value from square brackets
     * ANY Exception returns -1
     *
```

```
     * @param singularJSONInput a singular line of JSON [...]
     * @return a value between 1 and 100 or -1 on error
     */
    public static int extractIntFromSquareBrackets(String singularJSONIn
    {
        try
        {
            Pattern valuesWithinAnySquareBrackets = Pattern.compile("\\[
            Matcher matches = valuesWithinAnySquareBrackets.matcher(singu
            String value = matches.find() ? matches.group(1) : "-1";
            return  Integer.parseInt(value);
        }
        catch(Exception e)
        {
            return -1;
        }
    }

    /**
     * Gimmie' a 5! (of type int)
     *
     * @return      (int) 5
     */
    public static int doSomething()
    {
        return 5;
    }

    /**
     * ...
     *
     * @return If a random value between (1 : 100 + 5) > 54 then True els
     **/
    public static boolean doSomethingComplex()
    {
        String randomFromAPI = getSingleLineHttpResponse("http://www.rand
        int randomNumberFromAPI = extractIntFromSquareBrackets(randomFrom
        return ((randomNumberFromAPI + doSomething()) > 54);
    }

    /**
     * The imperative, unsafe (but useful) program!
     */
    public static void main(String[] args)
    {
        // Let the box heat up... can we know the result? NOPE.
        System.out.println(doSomethingComplex());
    }
```

```
}
```

We can observe the (somewhat predictable) unpredictable output of this code (yes, it does compile).


> *COFFEE TIME! You may remember me writing something about including 'coffee time' within some of my lecture notes, they're essentially exercises that I have thought about, that I might want YOU, the reader (if there are any) to think about.*
>
> *Question: Right, so we can all make out that this code will TYPICALLY return True or False, however, are there any other possible outcomes? What are they? Why might they occur?*

The point is: **arbitrary input output from any data source is possible anywhere within the programme, during execution**. The output coming from these sources may (and is very likely to) change, making it difficult to test. The further away we get from deterministic output (or, simply a small set of possible predictable outputs), the harder it becomes to test. When we're playing with money, **this is not a good thing!**

## 2.2 Now: For Some Haskell

So, you like the sound of declarative programming do you? **Good for you!** Personally, I am indifferent. I enjoy playing Snake on a Nokia 3310, but at the end of the day, you select the correct tool for the job. In my personal opinion (and I believe I can speak for the majority of the Cardano community here when I say): some kind of functional programming is the most appropriate tool when it comes to the implementation of smart contracts. Haskell is, as you well know, a functional programming language, so let's see an example.

**Haskell Example:**

```
foo :: int
foo = ...

let x = foo in ... x ... x ...
... foo ... foo
```

*Note: Haskell is a pure functional language*

*Now I'm going to assume you also are clued up on your basic philosophy:*

Premise 1: Given that the value of calling `foo` is unknown, and
Premise 2: Assuming foo has been assigned a return value[4]
Conclusion: Then foo will **always**[5] return the same value [1]

**Haskell knows that you're not a liar, so it keeps you to your word! This is explained in §2.3 as referential transparency.**

## 2.3 Referential Transparency

The following is an extract from a very well known Haskell tutorial / book:

> In purely functional programming you don't tell the computer what to do as such but rather you tell it what stuff is. The factorial of a number is the product of all the numbers from 1 to that number, the sum of a list of numbers is the first number plus the sum of all the other numbers, and so on. You express that in the form of functions. You also can't set a variable to something and then set it to something else later. If you say that a is 5, you can't say it's something else later because you just said it was 5. What are you, some kind of liar? So in purely functional languages, a function has no side-effects. The only thing a function can do is calculate something and return it as a result. At first, this seems kind of limiting but it actually has some very nice consequences: if a function is called twice with the same parameters, it's guaranteed to return the same result. That's called referential transparency and not only does it allow the compiler to reason about the program's behaviour, but it also allows you to easily deduce (and even prove) that a function is correct and then build more complex functions by gluing simple functions together. [1]

*Too Long Didn't Read?* Essentially, **referential transparency** is a property that does its best to maintain consistency in a value of an expression if the other components of the program **can change whilst maintaining** the value of such an expression.

## 2.4 Breaking Down The Haskell Example Some More...

> "The Hacker Way is an approach to building that involves continuous improvement and iteration. Hackers believe that something can always be better, and that nothing is ever complete."
> — Mark Zuckerberg

*I Believe that's a perfectionist Mr Zuckerberg, haven't you ever encountered a manager before? They're of a polar opposite nature. Anyway, this section only exists because I forgot where I was during the lecture notes, so feel free to skip over it!*

**Consider the following Haskell code:**

```
1. foo :: Int
2. foo = ...
3. ...
4. foo
5. ...
6. foo
```

To break it down outside of comments (since it's so simple): 1. initially we declare foo as type Int. 2. Then, we initialise foo to return a value of, let's say, 27 (for a laugh). 3. Right, so between lines 2. and 4. (Via the power of deduction: line 3!) some things happen... 4. Now, **no matter what happened prior, when we call foo**[6] (on line 4 and 6), it will always return the same value. **That value is: 27 of type Int and this property is known as referential transparency** [§2.3](#rt23). ### 2.5 Some More Referential Transparency As if we haven't covered enough. We're documenting everything here... Thus, consider the following.

```
1. let x = foo in ... x ... x ...
2. ... foo ... foo ...
```

During the first line, we're declaring assigning foo to x. Shortly thereafter, we're referencing foo via x. Then on line 2, we're also calling foo (there are intervals of 'other-worldly' program behaviour between these statements); but, due to referential transparency, we can be certain that (assuming foo still equals the value we set it to return) foo and x are going to provide the same value.

## 3. Input Output: Haskell Finally Has An Effect On The World!

*Why do we write programs?*

*Why do programmers exist?*

**These are some of the most profound questions oneself can ask oneself.**

Presupposition: Programmers exist in order to write programs (primarily). Premise 1: Programs must be useful. Premise 2: For programs to be useful, they must effect the world. Conclusion: only programs that can effect the world are useful.

> "Us geeky Haskell guys started with a completely useless language, in the end a program with no effect, there is no point in running it is there? You have this black box, you press go and it gets hot! There's no output, why did you run the program? The reason you run a program is to have an effect! But, nevertheless, we put up with that embarrassment for many years!" [2]
> — Simon Peyton Jones

**Finally!** We've managed to implement something in Haskell that **ACTUALLY DOES SOMETHNG!** How, you ask? Well, we use something called *the magic* of the **Type System**. Since Haskell is statically typed [4] since every expression in Haskell must have a type. Thus, it is possible to create a type constructor (for the compiler) that indicates that the type that it pre-fixes is to construct input and/or output (IO) of the post-fixed type. For example: `hello :: IO String` and `world :: IO Int` are of different types. This is because the `IO` *type constructor* is of type `IO String` with respect to `hello`. However, `world` has the type constructor `IO int`.

## 4. Type Constructors

See the following IO Type Constructor.

```
foobar :: IO Int
foobar = ...
```

The IO type constructor is described as by Lars as a 'recipe' to compute an Int and this computation can invoke side effects. Lars insists that it does not break referential transparency, so I can only assume that by recipe he essentially means: some code exists, which may be impure. However, the code / behaviour of the program outside of this impurity (arbitrary IO) itself never changes. Thus, the only thing that can change is the actual Input (or/and Output, depending on what the function that uses IO does). It must therefore follow that referential transparency is not broken.

---

## 5. More Info! Plus Some Code!

**Consider the following code, for a moment:**

```
import Data.Char
main :: IO ()
main = do
        putStrLn "Hey! What's your name?"
        name <- getLine
        let bigName = map toUpper name
        putStrLn $ "hey " ++ bigName ++ ", how are you?!"
```

**Now, let's get some stuff explained.**

- This above code is not being ran within a ghci.
- It's saved within a .hs (Haskell Extension) file.
- It's then being compiled using cabal `cabal run [SCRIPT_NAME].hs`.
- The reason for this is because is outlined within the next subsection.
- I would like to remind you that Haskell is Beautiful and **Pure**.
- Unfortunately, IO introduces a degree of terribleness into a beautifully pure language. Thus rendering it potentially impure. However, this does now mean our computer doesn't just turn into a radiator for those cold December months! *The one thing Haskell actually use to be good for!*
- Pure functions may be parameterless or they may not be. We can, however, know (for certain) that they will **always** return the same value (given that a parameterised pure function is provided with the same parameter[s]).
- Impure functions *spoil everything!* They introduce indeterminism into what would otherwise be deterministic (insofar as we can control any values passed to parameterised functions).
- Impure functions **do not always return the same value.**
- Impure functions **can only be implemented within impure functions...** So, I suppose we have to thank God (or Simon Peyton Jones) that the `main` main function (which is executed when the script is ran after having been compiled) is itself... Impure!
- You cannot implement impure functions within pure functions. If you do, prepare yourself for 'a domestic' with the compiler.

**Now that we've got 'some stuff' explained, let us break down the above script.**

- We need to be able to take input from the user, which typically comes as text.
- Text is of type string; and in Haskell String is the same as [Char].
- This means we need to import Data.Char.
- main :: IO () -- we're declaring the main function of our Haskell program & providing a function signature: The function is called `main` and it uses an IO type constructor of type (empty, nothing) unit.
- Being declarative about our use of functions is a **good thing** and it's **good practice**.
- Then we go on to define our function, main, which is the program itself (remember!?).

- The program is defined in terms of a `do` block.
- A `do` block is essentially the same as what you might use curly brackets for in a language like Java.
- Next line is simple, output the text proceeding the function name to console.
- Again, the next line simply takes the input from the console and assigns it to `name`.
- Now, we're going to assign the result of the `map` function to `name`.
- map takes two parameters, the first is (in this case) a function itself: toUpper, which - as you have might guessed: takes every character of a string and converts it to an upper case character. The second parameter is the result of the input provided by the user stored within name, which was obtained using getLine (a function Haskell implements itself).
- Finally, all we're doing is outputting to console the concatenation of three strings (or [Char]s).
- All in all, a very simple program; but for anyone not use to programming in Haskell, it's good to run through this stuff.

## 5.2 So, Why Are We Running This With Cabal?

Recall that there may be additional packages that we require in order to compile and consequently run the Haskell we are writing. Why? Well, because there may be elements of Plutus that we require (although that isn't necessarily the case with this small script) in order to allow our off-chain code (and when we compile on-chain code, we will definitely need those additional packages!) to interface with the Plutus Backend. It should be fairly self-evident at this point, but that is the reason for package management, see the `.devcontainer` that cabal is using for this exercise below.

```
{
    "name": "Plutus Starter Project",
    "image": "plutus-devcontainer:latest",

    "remoteUser": "plutus",

    "mounts": [
        // This shares cabal's remote repository state with the host. We
        // 1. '.cabal/config' contains absolute paths that will only mak
        // 2. '.cabal/store' is not necessarily portable to different ve
        "source=${localEnv:HOME}/.cabal/packages,target=/home/plutus/.ca
    ],

    "settings": {
        // Note: don't change from bash so it runs .bashrc
        "terminal.integrated.shell.linux": "/bin/bash"
    },

    // IDs of extensions inside container
    "extensions": [
        "haskell.haskell"
    ],
}
```

So, this `.devcontainer` is mounting our cached cabal packages from our Plutus repo; and it's doing this as we've built the cabal project file for this week, which contains references to the dependencies within Plutus, in addition to the execution paths for any programs we may need to run. See below.

```
Cabal-Version:       2.4
Name:                plutus-pioneer-program-week03
Version:             0.1.0.0
Author:              Lars Bruenjes
Maintainer:          brunjlar@gmail.com
Build-Type:          Simple
Copyright:           © 2021 Lars Bruenjes
License:             Apache-2.0
License-files:       LICENSE

library
  hs-source-dirs:      src
  exposed-modules:     Week03.Homework1
                     , Week03.Homework2
                     , Week03.Parameterized
                     , Week03.Solution1
                     , Week03.Solution2
                     , Week03.Vesting
  build-depends:       aeson
                     , base ^>=4.14.1.0
                     , containers
                     , data-default
                     , playground-common
                     , plutus-contract
                     , plutus-ledger
                     , plutus-ledger-api
                     , plutus-tx-plugin
                     , plutus-tx
                     , text
  default-language:    Haskell2010
  ghc-options:         -Wall -fobject-code -fno-ignore-interface-pragmas
```

*Note: I will need to revisit this portion of the lecture notes and potentially brush up on them... But, I think I'm doing OKAY...*

## 6. Basic: Function: Map & toUpper & toLower, IO String, IO [Char]

In Haskell [Char] is a list of characters, which equates to a string. So, when you: `map toUpper` "a lower case string" within the repl, you'll see "A LOWER CASE STRING" returned. A similar function exists called toLower, I won't go into that - I think you can guess what it does.

## 7. Concept: Functor, fmap | Getting Slightly More Complicated

- Important in Haskell

- fmap :: (a -> b) -> f a -> f b
- This essentially reads: there exists a function called fmap
- I imagine this stands for function map or function mapping
- It takes a parameter a, which it then passes to b
- This means it can take something like the function `map` and `toUpper`, then you can innovate the function `getLine` which results in a mapping from IO (input from the user / console) to the toUpper function, converting any string input to upper case (the mapping of one functions output to another's input, resulting in a final output). Thus, turning one IO actions into another's IO actions (embedded impure functions).

**EXAMPLE:**

We are only interested in the case where f is IO. Thus, if we use fmap, in combinatin to map toUpper... Well, you can guess what happens! Remember `String = [Char]`. This type of usage of `fmap` is typically used for (in our case) transforming user input after it has been entered into, say, the console.

```
fmap :: (a -> b) -> f a -> f b
x = fmap (map toUpper) getLine
putStrLn x
```

**IO Chaining:**

Okay, so we've seen how we are able to take user input and transform it whilst maintaining the referential transparency. What we have no yet encountered is essentially a double `>>`. This is a chaining operation called **sequence to operator**. Recall that putStrLn has a unit result `()` and so if we chain two putStrLn functions together:

```
putStrLn "Learning" >> putStrLn "Haskell"
```

The LHS function is executed first *(note: as the result is simply (), nothing special happens anyway, but if a type result existed then the >> operator would simply ignore it)*, then the RHS is executed. In short: given two *'recipes'* the `>>` operator will executed LHS, throw away the result, then execute RHS & attaches the second recipe.

## 8. Concept: Chain Binding | Important: The RHS Result Is Not Ignored

**Warning: `:t (>>=)` will start talking about a Monad constraint if executed in the repl, but for now, let's just worry about the IO.**

```
-- written like: >>= | this does not ignore the first result
:t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

If a recipe exists that performs side effects and obtains an 'IO a', then that 'a' is passed to a function where: given 'a' it provides a function that returns 'IO b' then these two can be combined such that a new recipe exists that gives b. Essentially mapping the output from the first IO into the second function via 'bind chaining' - For example:

```
getLine >>= putStrLn
```

Will wait for user input, and instead of simply throwing it away (as is the case with simple IO chaining), it feeds the IO into the second function, which will output it to console:

```
Plutus ... > getLine >>= putStrLn
HelloWorldIO
HelloWorldIO
Plutus ... >
```

**Yet Another Haskell Program (To Be Compiled):**

```
main :: IO ()
main = foo

foo :: IO ()xs
foo = getLine >>= \t ->
      getLine >>= \s ->
      putStrLn (s ++ t)
```

*Note: You can rewrite this using* `do` *blocks.*

```
main :: IO ()
main = do
       getLine >>= \t ->
       getLine >>= \s ->
       putStrLn (s ++ t)
```

## 9. Function: Return

```
Plutus ... > return "Haskell" :: IO String
"Haskell"
```

## 10. Characters and Strings

Characters are individual quotes, strings are double quotes. However, a list of characters is essentially a string:

`IO String === IO [Char]`.

## 11. Concatenation using Semicolon (Careful!)

```
Plutus ... > 'A' : " little dog"
"A little dog"
```

*Has to be a character followed by a string*

## 13.1 Maybe.hs

Maybe is a great way to handle IO once it has entered into the program. The purpose of Maybe is to pre-fix the term to a return type. In short, it's essentially saying: in an ideal world, you would get the following return type, but since we do not live in a utopian paradise, you may not get what you're looking for. Hence: Maybe.

```
foo :: String -> String -> String -> Maybe Int
```

Upon defining `foo` and allowing it to execute, one should notice that if any type submitted as a parameter to the function is not a simple value wrapped in a string, the function will return `Nothing`, which is essentially an exception in Haskell. However, if all three values are Integers wrapped within Strings, it will return the summation of the parameters.

### Maybe.hs Implementation & My Own Personal Comments

*Please Keep The Following In Consideration Before Rendering Judgement.*

Firstly, I am in no way anywhere near as qualified as any of the individuals developing the majority of Haskell applications for production environments. I am, simply, a student of Haskell (hopefully, not for long — I believe I pick things up fairly quickly). Some additional observations:

- I believe Haskell programmers air more so on the side of mathematicians than most software engineers, probably because most software engineers are using imperative OOP, not declarative functional.
- Due to the above observation, Haskell programmers tend to use single letters to assign values to variables or during function declaration. This is in opposition to what 'conventional wisdom' from 'Uncle Bob' [REF] would teach us: the process of reading code is similar to what you might consider an identity function `f(x) = x`, essentially, you shouldn't really need to comment code because 'good' code should be readable and understandable.

- This is why I have occasionally been somewhat skeptical of programmatic 'magic' - you'll see what I see in a second.
- In regards to the implementation of Maybe.hs and Monad.hs (week four); whilst the first instance of reading three strings as potential Integers is, admittedly, a little ugly and, yes, there is a principal called DRY (Don't Repeat Yourself). However, at the same time there is another principal called KISS (Keep It Simple Stupid).
- Yes, whilst the implementation of foo is ugly, it is simple and easy to understand.
- The use of bindMaybe and the implementation of foo' is a little nicer, but requires a small amount of basic Haskell knowledge (no issues with this really).
- Now, whilst it's expressed somewhat eloquently in two lines of code, the implementation of foo'' requires a little bit more thought when deconstructing what is actually being expressed. For me, it's a little like one of those moments where you're a bit like: 'hang on, what exactly is going on here...?' — it reminds me of being a junior developer and encountering dependency injection (in Laravel) for the first time. Upon this 'discovery' a mid-level developer told me: *don't worry about it, it's just "Laravel Magic"* — not exactly the explanation you are looking for as an inspiring developer!
- The crux of the argument boils down to the following: is it better to write somewhat verbose code that is self-explanatory, or does it make more sense to abstract complexity away from the 'main thread' (as it were). For those who understand what is going on (because they have contributed to building the abstraction mechanisms, it is fantastic — because it minimises amount of required code to read, which is what we spend most our time doing. However, for those who approach it without having had the privilege of being enrolled in something like the pioneer program, it may hinder (their) progress.
- The last thing I would want is to hinder the magic involved in learning something new due to frustration.
- For this reason, I, myself, will always try my very best to take the time to explain these things to people.
- This section of the lecture notes are purely academic and philosophical masturbation (really, that is what I'm doing here). But, these are interesting questions to think about and raise.
- My personal opinion: Blockchains and Distributed Ledgers is a multi-disciplinary, novel subject. It requires (arguably) a postgraduate level of education to jump into it fairly quickly, unless one is incredibly bright. With this in mind, I do believe foo'' was the most appropriate implementation, but for a junior level developer, it may be a bit much. They would likely prefer the implementation of foo.

*My 2 cents.*

## 3.2 Maybe.hs

```
module Week04.Maybe where

import Text.Read (readMaybe)
import Week04.Monad

foo :: String -> String -> String -> Maybe Int
foo x y z = case readMaybe x of
    Nothing -> Nothing
    Just k  -> case readMaybe y of
        Nothing -> Nothing
        Just l  -> case readMaybe z of
            Nothing -> Nothing
            Just m  -> Just (k + l + m)

bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindMaybe Nothing  _ = Nothing
bindMaybe (Just x) f = f x

foo' :: String -> String -> String -> Maybe Int
foo' x y z = readMaybe x `bindMaybe` \k ->
             readMaybe y `bindMaybe` \l ->
             readMaybe z `bindMaybe` \m ->
             Just (k + l + m)

foo'' :: String -> String -> String -> Maybe Int
foo'' x y z = threeInts (readMaybe x) (readMaybe y) (readMaybe z)
```

## 13.3 Monad.hs

```
module Week04.Monad where

-- (>>=)      :: IO a            -> (a -> IO b)            -> IO b
-- bindMaybe  :: Maybe a         -> (a -> Maybe b)         -> Maybe b
-- bindEither :: Either String a -> (a -> Either String b) -> Either S
tring b
-- bindWriter :: Writer a        -> (a -> Writer b)        -> Writer b
--
-- return              :: a -> IO a
-- Just                :: a -> Maybe a
-- Right               :: a -> Either String a
-- (\a -> Writer a []) :: a -> Writer a

threeInts :: Monad m => m Int -> m Int -> m Int -> m Int
threeInts mx my mz =
    mx >>= \k ->
    my >>= \l ->
    mz >>= \m ->
    let s = k + l + m in return s

threeInts' :: Monad m => m Int -> m Int -> m Int -> m Int
threeInts' mx my mz = do
    k <- mx
    l <- my
    m <- mz
    let s = k + l + m
    return s
```

## 13.4 Either

Either is a function that would be used / called when you're looking to implement code that (when it breaks) returns helpful error messages. Although, it could be used for all kinds of things, that is the use case we are specifically looking at.

- Two Type Constructors as parameters, for example, the type constructors: `Left` and `Right`.
- One Return Type
- `type Either :: * -> * -> *`
- `data Either a b = Left a | Right b`
- Problem with Maybe: it returns `nothing`, so there is no error message.
- RHS: Type: Just
- LHS: Type: (some kind of error), you could implement this as a String.

```
module week04.Either where

import Text.Read (readMaybe)

readEither :: Read a => String -> String a
readEither s = case readMaybe s of:
    Nothing -> Left $ "Error! Cannot Parse: " ++ s
    Just a  -> Right a
```

Now, it becomes possible to implement our code previously written for `Text.Read` by replacing `Maybe` with `Either` and `readMaybe` with `readEither`.

*Note: We also need to replace the 'Nothing Exceptions' and the Just ... to Left and Right return values respectively. Furthermore, we need to adjust the next portion of code too, in order to reflect our changes*

```
bindEither :: Either String a -> (a -> Either String b) -> Either Stri
ng b
bindEither (Left err) _ = Left err
bindEither (Right x) f = f x

foo' :: String -> String -> String -> Either String Int
foo' x y z = readEither x `bindEither` \k ->
             readEither y `bindEither` \s ->
             readEither z `bindEither` \m ->
             Right (k + l + m)
```

*Note: At this point I simply watched the remainder of the lecture. I may come back and add more notes when there is more time available. For now, I need to get Lecture Five and Six written up & move on to lecture 7 ASAP. I have to say though, this has been the most difficult lecture yet. Five and Six are not that bad, well, I'm half way through six.*

*Note: Lars was no kidding when he said this is not easy... I'm pretty use to working A LOT, but moving to declarative functional is very strange. If you don't really know much Haskell, or worse, if you've never come across it in your life, you'll be required to put in 50 to 60 hour weeks to get this course done IMO - about 30 hours of Haskell and 20 - 30 hours of Plutus.*

---

## Images

```
True
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> :i IO
type IO :: * -> *
newtype IO a
  = ghc-prim-0.6.1:GHC.Types.IO (ghc-prim-0.6.1:GHC.Prim.State#
                                   ghc-prim-0.6.1:GHC.Prim.RealWorld
                                 -> (# ghc-prim-0.6.1:GHC.Prim.State#
                                         ghc-prim-0.6.1:GHC.Prim.RealWorld,
                                       a #))
        -- Defined in ‘ghc-prim-0.6.1:GHC.Types’
instance Applicative IO -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Monad IO -- Defined in ‘GHC.Base’
instance Monoid a => Monoid (IO a) -- Defined in ‘GHC.Base’
instance Semigroup a => Semigroup (IO a) -- Defined in ‘GHC.Base’
instance MonadFail IO -- Defined in ‘Control.Monad.Fail’
Prelude Plutus.V1.Ledger.Interval Week03.Homework1>
```

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> import Data.Char
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1> fmap (map toLower) getLine
HELLO
"hello"
```

```
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1> :t fmap (map toUpper) getLine
fmap (map toUpper) getLine :: IO [Char]
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1>
```

```
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1>
```

```
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1> return "Haskell" :: IO String
"Haskell"
Prelude Plutus.V1.Ledger.Interval Data.Char Week03.Homework1>
```

## References


[1] Lipovaca, M., 2011. Learn you a haskell for great good!: a beginner's guide. no starch press.

[2] Haskell is useless. Simon Peyton Jones. December 18, 2011. https://youtu.be/iSmkqocn0oQ


## Footnotes

1. Which, for most of us, we know all too well... For those who maybe are stuck in the ancient past and are still using punch cards: these are languages which essentially say: do

this, now do this, now do this (sounds more like procedural, but, not to go down a rabbit hole here, procedural programming is simply a subset of imperative programming); Procedural is typically thought of as similar to assembly (do this, now do this, now go here, now do that), where as object-orientated (still imperative) has a more elegant (and possible eloquent) way of describing, defining and maintaining state. However, we do also have declarative programming languages, functional programming falls into this camp, and thus, Haskell is a declarative programming language.

2. Making them fairly difficult to test to the same level of scrutiny as say something such as a mathematical function[3]. Since Haskell is a functional programming language, this makes Haskell pretty darn easy to test. Thus, pretty darn safe (once again, thank you Simon [2]).

3. To my limited mathematical knowledge: A mathematical function is defined (almost as though it is some kind of constant) as having an input and facilitating an output. A 'fruity' question I had to ask myself was: how exactly can you implement a mathematical, functional programming language within a discrete system? The obvious answer being: discrete mathematics... However, now we have to apply a **whole bunch** of constraints to such a language, which is probably why it's so safe? At least, perhaps one reason why? These are my notes, so take them for what they're worth, which may be absolutely nothing.

4. A function does not need to do any arbitrary IO, but functions tend to return something, even if it's just unit data: `()`.

5. Given a function f(x) = x ... **Given input x, the output is deterministic**.

6. Although there may be some edge cases that we're not going to worry about for now about how we call foo which ***may*** change its value.