Haskell: A Fun, Friendly, Fantastic Functional Language

Andrew Brinker

Hello,

Welcome to the wonderful world of Haskell. Haskell is my personal favorite programming language, and is the most exciting and interesting language I've ever encountered. It sometimes has a reputation for being academic or math-y. In this introductory workshop I will attempt to show you just how untrue that is. By the end you should be able to make simple (but useful!) programs using Haskell, and you should have a basic understanding of the techniques used to do so. If at any point you have a question about something I've said. Please don't hesitate to ask!

Thank you, Andrew Brinker

What is Haskell?

Before we get into how to actually build stuff in Haskell, you should have at least a basic understanding of what Haskell is, and how it differs from the programming languages you're likely to already know.

Haskell is a lazy, purely functional programming language. This means a few things:

Laziness: Last-Minute Evaluation

One of the ways that Haskell improves the performance and expressiveness of code is through **laziness**. This means that nothing you write in Haskell is evaluated until it has to be. Let's look at an example from GHCi, the Haskell interpreter:

```
> let x = 3 + 7
> x
10
```

You might think that the `3 + 7` expression is evaluated in the first line, but it's not. In fact, evaluation doesn't happen until the second line. The first line is instead a promise to evaluate x to that particular value when it's needed. Until x is called for, the evaluation doesn't happen, and if x is never called for, the *evaluation never happens at all*.

This is laziness, and while it may seem odd, it actually allows for some pretty cool things. Lets say you want to define a list of all odd integers.

```
> let odd = [a | a <- [1..], a `mod` 2 == 1]
```

Don't worry about the syntax right now. Just know that this lazily creates a list of every single odd number, to be evaluated one-by-one whenever you need it. In other languages, this would be impossible, because everything is evaluated immediately, and defining a list of infinite items will never halt. In Haskell, so long as you don't do any operations that attempt to get the whole list (like taking the sum of all its members, for example), you're fine. So, if you want the first ten odd numbers:

```
> take 10 odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Viola! It works!

Pure Functional Programming: A New Paradigm

Functional programming is a programming paradigm focused on the application of functions to inputs and the avoidance of state. This means that instead of stating **how** something is done, you instead describe **what** should be done.

Let's look at an example. To take the sum of all the values in a list of integers, you might do this:

```
int sum(std::list<int> l) {
    int s = 0;
    std::list<int>::iterator it = l.begin();
    while (it != l.end()) {
        s += *it;
        ++it;
    }
    return s;
}
```

This describes how to take the sum, which in this case is done by iterating through the elements in the list, at each turn adding to a running total. Let's look at the same function in Haskell:

```
sum :: (Num a) => [a] -> a
sum [] = 0
sum (head:rest) = head + sum rest
```

This instead defines the function recursively, saying that the sum of a list is the value of the first element plus the sum of the rest of the list. It performs the same exact task as the C++ function, but doesn't actually explain how the iteration through the list should be done. Instead, it defines an almost obvious truth about sums, which turns out to be completely valid code!

But what does it mean to be "purely functional"? Well, in this context "pure" just means that the function returns the same output for a certain input *no matter what*. If you write a function like this:

```haskell
addOne :: (Num a) => a -> a
addOne x = x + 1
```

And call it like this:

```haskell
addOne 5
```

It will always give the same result (6). This may seem stupid, but imagine a function in C++ that claims to do the same thing:

```cpp
int addOne(int x) {
    std::ifstream input_file("blah.txt");
    if (input_file.good()) {
        return x + 2;
    }
    return x + 1;
}
```

In this case, the function tries to open a file. If the file exists and is opened successfully, the function returns `x + 2` instead of `x + 1`. And while it's unlikely that anyone would write such a stupid function, there's nothing in the language to stop them from doing it, or tell you they've done it without you cracking open the code and reading it.

In Haskell, trying to do the same thing (opening a file and optionally returning `x + 2`) is impossible. Simply adding file handling to the function changes the function entirely, so that it can't be called the same way. Essentially, the compiler forces the programmer to make sure that the function is *referentially transparent*, or 'pure' so that it always behaves predictably.

What It Is

So now, when I say that Haskell is a lazy, purely functional programming language, you hopefully have a better appreciation of what that means. If not, let's continue anyway. Hopefully some more examples make it clear.

Types, Types, Types

There are few things in Haskell more important than **types**. I'm sure you're familiar with the idea of types, like `int` or `bool` or `char` in C++. They provide a way for the compiler to make sure that the code you're writing makes sense (it wouldn't make sense, for example, to write `2 + true`). Put another way, they allow the compiler to make sure your program is behaving according to some expected static semantics (or, compile-time meaning).

Types in Haskell are a lot stronger and cooler than types in C++ (please C++, don't be mad at me for saying so. I still sorta like you). First of all, you don't have to say them, the compiler automatically figures them out (except in rare cases where it can't). Second, in Haskell functions are just another type of variable, which lets you do things like pass functions to functions and return functions from functions, all in a

type-safe, compiler-checked way. Third, thanks to typeclasses (which we'll cover later), you can write generic functions which accept a variety of input types, and still guarantee that your code is safe.

In general, all of this awesomeness means that it can be a little harder to get Haskell code to successfully compile than it is in other languages, but it also means that once your code compiles you can feel pretty sure that it is correct. Put another way, Haskell takes errors that would otherwise be runtime errors, and moves them to compile time (trust me, this is a lot better).

The basic types in Haskell are:

- `Int`: The basic integer type, whose size is based on the current machine.
- `Integer`: Arbitrary size integer, which can grow (theoretically) infinitely.
- `Float`: 32-bit floating point number, as defined by the IEEE
- `Double`: 64-bit floating point number, also defined by the IEEE
- `Bool`: True or False. Not much more to say.
- `Char`: A Unicode text character.

These types should seem relatively familiar, as they are roughly analogous to the basic types in C and C++. To get a better feel for them, let's fire up GHCi (the Glasgow Haskell Compiler interpreter) with the `ghci` command.

Each type has certain operations that can be performed on it. For example, all numeric types can be added, substracted, multiplied, and divided, like so:

```
> 3 + 5
8
> 3.5 - 4.2
-0.7000000000000002
> 4 * 9
36
> 10 / 2
5
```

Boolean types can be ANDed and ORed and all that:

```
> True && False
False
> False && not True
False
> False || True
True
```

Strings (defined as lists of characters), can be concatenated:

```
> "Hello, " ++ " world!"
"Hello, world!"
```

In every case, these operations are essentially what you would expect, except that these operators are nothing more than nice looking syntax on top of Haskell's normal function system. Here is some math, written a different way:

```
> (+) 5 2
7
```

In this case, `(+)` is the function name (the parentheses make Haskell treat it as a prefix operator, rather than the usual infix one), and 5 and 2 are the parameters. Notice than in Haskell there are no parentheses or commas needed to pass something into a function.

Let's try some operations with incorrect types and see what happens:

```
> 2 + True
<interactive>:2:3:
    No instance for (Num Bool) arising from a use of '+'
    In the expression: 2 + True
    In an equation for 'it': it = 2 + True
```

We'll get to what `(Num Bool)` means soon, but for now just notice that the compiler immediately saw the operation was invalid, and provided a nice error message explaining exactly what went wrong. This is a wonderful thing, and something that Haskell is very good at.

There are two more types I want to talk about: lists and tuples. Together they form the basis for much of what you're likely to do in Haskell, and we'll be using both throughout the rest of this workshop.

Lists are homogeneous data structures. Every item in a list is of the same type. In Haskell, lists look like this:

```
> [1,2,3]
```

And their type is this:

```
[a] -- Where 'a' is the type of whatever thing is inside.
```

All Strings in Haskell are actually lists of characters (`[Char]`). Furthermore, all lists are actually constructed like this:

```
1 : 2 : 3 : []
```

The `:` operator you see here is called the "prepend" operator, and it takes the first argument and puts at the front of the list in the second argument. So, solving from the right, the above becomes:

```
1 : 2 : 3 : []
1 : 2 : [3]
1 : [2, 3]
[1, 2, 3]
```

All lists in Haskell can be constructed this way, and you can use this to easily work with lists in a variety of ways.

Lists can also be constructed using the magic of **list comprehensions**. They look like this:

```
let odd = [a | a <- [1..], a `mod` 2 /= 0]
```

This is the same list of odd numbers from before, and it's a list comprehension! The generic syntax for list comprehensions is:

```
[<variable(s)> | <source>, <condition>]
```

In the odd numbers example, the list comprehension can be read as: the list of all a, such that a is a positive integer that is not divisible by 2. As far as the source goes, it means a is drawn from the infinite list of integers starting with 1.

Now tuples aren't like lists. They are instead heterogenous structures of a finite size. Here is an example:

```
> (5, "Hello")
```

This is a tuple of type `(Int, [Char])`. Tuples are great for doing things like returning multiple values from a function (you put them together in a tuple), like this:

```
what :: Int -> Int -> (Int, Int)
what a b = (a - 5, b ^ 2)
```

This function returns a tuple based on some numeric operations on the input values. It's not particularly useful, but it illustrates the idea.

Functions

So, we've talked a bit about the basic types, now it's time to talk about functions. Given that Haskell is a _function_al programming language, you can rest assured that functions are extremely important for writing good, real-world Haskell programs. Let's start with a simple example of what a Haskell function looks like:

```
addOne :: Int -> Int
addOne x = x + 1
```

You say this function earlier (albeit in a slightly different form), but we didn't actually talk about what's going on here. Let's walk through it. The first line is the *type annotation*. It tells the Haskell compiler what the type of this particular function is. In this case, `addOne` is a function which takes in an integer and returns an integer. Makes sense, right?

The next line is the actual implementation of the function. It says the function name again, and then lists the parameter (just `x` in this case). Then it has an equal sign, and the actual function body.

To better illustrate some of what's happening here, let's see another one:

```
add :: Int -> Int -> Int
add x y = x + y
```

In this case, we have two parameters, x and y, and both are integers. If you look at the type annotation though, you'll notice that the syntax for one parameter and a return value is the same as the one for two and a return value. In fact, the only thing that indicates which one is the return value is the fact that it comes last!

In reality, all functions in Haskell actually take *only one parameter*. When you write something like the function add above, you're actually writing something more like this:

```
add :: Int -> (Int -> Int)
add x = \y -> x + y
```

This is a function that takes in a single parameter x, and then returns another function that takes a parameter y, which then returns a single integer value.

This process is called **currying** (named after logician Haskell Curry, for whom Haskell is named), and allows you to do some pretty cool things, like this:

```
> let add x y = x + y
> let addTen = add 10
> addTen 5
15
```

In this example (which is back in the Haskell interpreter), we remade our add function without the type annotation (which is optional). Then we made a new function by passing only a single parameter to add. This function takes in a single parameter of its own, and then adds 10 to it! Just like that, we took a generic function we had and made a new one like it was nothing. That's the power of currying.

That's not the only cool thing you can do with functions. Take a look at this:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

What's this? This is a classic function called `map`, which is used all the time in functional programming. Here's an example of it in use:

```
> map (* 2) [1,2,3,4]
[2,4,6,8]
```

Basicaly, it takes a function (like `(* 2)`) and applies it one-by-one to each item in a list (like `[1,2,3,4]`).

What? It takes a function as a parameter? Yes. It does. Functions in Haskell are just like anything else. They have a type, and they can be passed around to eachother however you like. Let's look at that `map` implementation again:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The body is doing some stuff we haven't covered yet, but the type annotation is pretty straightforward. It takes in two parameters, one is a function from type `a` (which can be anything) to type `b` (which can also be anything), the other is a list of values of type `a`. It then converts those values into values of type `b`.

In the particular case we showed, `a` and `b` are both `Int`. But they don't have to be, and they don't have to be the same thing. Here's another example:

```
> map show [1,2,3,4]
["1","2","3","4"]
```

This time, the `show` function converted each of the numbers into strings. So `a` is `Int`, but `b` is `String`.

Learning to read type annotations is important in Haskell, and we'll be practicing it throughout these lessons. If you don't quite understand what we've already covered, feel free to go through it again.

GHC: The Glasgow Haskell Compiler

We've covered a lot already. Before continuing on, let's go through the steps of making a simple Haskell program and compiling it using the Glasgow Haskell Compiler. You'll need these steps in the upcoming sections.

First, Haskell programs are defined in files with the `.hs` extension. So, to start a new Haskell file, type something like `touch main.hs`. This will create your empty file to begin editing. Next open it up in your favorite text editor, and type this:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

This is just a basic Hello World program, like you would see in any other language. Save the file, and then compile it with `ghc --make main.hs`. You can then run the generated file with `./main`. Run it, and make sure it prints out `Hello, World!`.

That's how you use the Glasgow Haskell Compiler. We'll be using it throughout the remainder of the workshop, so make sure it's running correctly and your code compiled and ran successfully. If it didn't, ask for help.

Pattern Matching

Let's look at that `map` function we defined earlier:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

This uses something called **pattern matching**. Basically, the function defines two unique patterns, and adjusts its behavior accordingly. The first pattern says that any function mapped over an empty list results in an empty list. The second one says that any function mapped over a list with at least one element is the same as the result of applying the function to the first element, and then mapping it over the rest.

Let's look at how that would work in this case:

```
> map (* 2) [1,2,3]
```

We already said what would happen from running this code, but let's actually walk through it:

```
map (* 2) [1,2,3]                          -- Pattern: map f (x:xs)
(* 2) 1 : map (* 2) [2,3]                  -- Pattern: map f (x:xs)
(* 2) 1 : (* 2) 2 : map (* 2) [3]          -- Pattern: map f (x:xs)
(* 2) 1 : (* 2) 2 : (* 2) 3 : map (* 2) []  -- Pattern: map _ []
(* 2) 1 : (* 2) 2 : (* 2) 3 : []           -- Eval: map (* 2) [] -> []
(* 2) 1 : (* 2) 2 : 6 : []                 -- Eval: (* 2) 3      -> 6
(* 2) 1 : 4 : 6 : []                       -- Eval: (* 2) 2      -> 4
2 : 4 : 6 : []                             -- Eval: (* 2) 1      -> 1
2 : 4 : [6]                                -- Eval: 6 : []       -> []
2 : [4,6]                                  -- Eval: 4 : [6]      -> [4,6]
[2,4,6]                                    -- Eval: 2 : [4,6]    -> [2,4,6]
```

This is what is looks like when the initial code `map (* 2) [1,2,3]` is evaluated. At each stage, the map function matches the current input against the defined patterns, and runs the code accordingly. Once `map` application is done, the multiplication begins to happen, following by the concatenation. In the end, you have a list that is simply the original one with each element multiplied by 2.

Functions can pattern match in many ways. Here is another example:

```
lucky :: Int -> IO ()
lucky 5 = putStrLn "Winner!"
lucky _ = putStrLn "Meh"
```

In this case, the function `lucky` will print "Winner!" when the input is 5, and "Meh" the rest of the time. It does this by matching the input against a particular value. This works just the same as the last example did.

Guards

When you want to text certain conditionals in Haskell, you can use guards, which are similar to pattern matching. Here is an example:

```
biggerest :: Int -> Int -> Int
biggerest a b
  | a >= b    = a
  | otherwise = b
```

This function will return the "biggerest" of two numbers, and it uses guards to do it. In this case, you test conditionals (like you would in C++ with if and else), and execute particular code accordingly.

You can also temporarily bind variables in guards as well, like so:

```
bmiTell :: Float -> Float -> IO ()
bmiTell weight height
    | bmi <= 18.5 = putStrLn "You're a bit under your healthy weight."
    | bmi <= 25.0 = putStrLn "You're at a healthy weight."
    | bmi <= 30.0 = putStrLn "You're treading into unhealthy territory."
    | otherwise   = putStrLn "You have a seriously unhealthy BMI."
    where bmi = weight / height ^ 2
```

In this case, the function temporarily defines the variable `bmi`, which is more like a constant because it can't be reassigned, and uses it in the guard conditionals to avoid having to write (or do) the calculation multiple times.

Let

You can also declare variables elsewhere using `let` clauses, like so:

```
hello :: String -> String
hello name =
    let phrase = "Hello "
    in  phrase ++ name
```

Of course, this is a trivial example, but it shows what I'm talking about. `let` is used to instantiate variables. It can also be used to instantiate functions, like so:

```
> [let square x = x * x in (square 5, square 3, square 2)]
[(25, 9, 4)]
```

This uses the square function to then create a list containing a single tuple of three numbers.

`let`, like most of Haskell, is very flexible, and can be used in a variety of contexts. Experiment with it, and see what it can or cannot do.

Lambdas

Next up is Lambdas, which are just anonymous functions, and are written like this:

```
> map (\x -> x^3) [1,2,3]
[1,8,27]
```

This created a function without a name, and it used a special syntax to do it. With lambdas, you start with the backslack, and then list the parameters, and finished it off with the arrow and function body. Lambdas are really only used in places where it's not worth it to define a separate function, and they otherwise behave identically to functions.

Typeclasses

Until this point I've been dancing around something important. In fact, it's one of the most important topics in Haskell, and it's something you're definitely going to need if you're going to create real world Haskell programs. It's the wonderful world of **typeclasses**.

To quote "Learn You A Haskell For Great Good" (a fun, free introductory book for Haskell):

> *A typeclass is a sort of interface that defines some behavior. If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes. A lot of people coming from OOP get confused by typeclasses because they think they are like classes in object oriented languages. Well, they're not. You can think of them kind of as Java interfaces, only better.*

Everything in Haskell uses typeclasses. In fact, when I talked about types near the beginning of this whole thing, I was sort of lying. Let's take a look at what we get when we check the type of the number 5 in GHCi:

```
> :t 5       -- Note, ':t' exists only in GHCi, not in the Haskell language
5 :: Num a => a
```

You would expect 5 to be an `Int`, but it isn't. Instead it's a `Num a`. What's that? That's a typeclass!

One of the wonderful things in Haskell is its laziness. That includes not deciding on the type of a value until it has to. In this case, Haskell doesn't immediately classify 5 as an `Int`. Instead, it keeps it as some sort of generic number, and can then turn it into any of the classes that derive from the `Num` typeclass. Haskell will actually keep every value as the most general type it can until the last minute. This makes your programs flexible.

You actually write functions using typeclasses. Let's take a look at `addOne` again:

```
addOne :: Num a => a -> a
addOne x = x + 1
```

Now, instead of specifying `Int` as the input type, I used `Num a`. So, anything that derives the `Num` typeclass can be passed into the function. Let's see that in action:

```
> addOne 5
5
> addOne 5.0
6.0
```

Look at that! It worked for both an integer and a floating point number, all thanks to the wonder of typeclasses.

There are actually a number of typeclasses commonly defined and used in Haskell. They are:

- `Eq`: The typeclass for equality comparison
- `Ord`: The typeclass for things that can be ordered sequentially

- `Show`: The typeclass for stuff that can be turned into strings
- `Read`: The typeclass for stuff that can be derived from strings
- `Enum`: The typeclass for things which can be enumerated
- `Bounded`: The typeclass for things with an upper and lower bound
- `Num`: The typeclass for numbers
- `Integral`: The typeclass for integers (`Int` or `Integer`)
- `Floating`: The typeclass for floating point numbers (`Float` or `Double`)

There are more typeclasses in Haskell (many many more), but these are the basic ones.

Algebraic Data Types

The final thing to know before we finally make a real world Haskell program is how to define your very own custom types. With all the talk of types thus far, I'm sure you're aware of the importance of type definitions in Haskell. The ability to define your own types is one of the language's key features, and it's something that is absolutely integral to the creation of any substantive Haskell program. Here is what it looks like:

```
data Bool = False | True
```

That is the actual in-language definition for `Bool`! Literally, a Bool is either `True` or `False`. Those two are called *type constructors*. Literally, they are functions that construct values of type `Bool`. Let's take a look at another example:

```
type Point = Float Float
type Length = Float
type Radius = Length

data Shape = Circle Point Radius
           | Rectangle Point Length Length
```

In this example we've done a few new things. First of all, we've declared **type aliases** using the `type` keyword. Essentially, we've defined new types that are equal to come combination of existing types. These are generally done to make your types more expressive about what they represent, and to improve type checking (the compiler will now treat `Length` and `Float` as different, for example, even though they are syntactically the same, because they represent different ideas semantically).

Second, we've given our type constructors `Circle` and `Rectangle` parameters! So, if we wanted to define a circle, we could do something like this:

```
> let c = Circle (Point 5.0 5.0) 1.0
> :t c
c :: Shape
```

This constructs a new `Circle` by first constructing a `Point` and then passing it to the `Circle` constructor. The created variable is, as expected, of type `Shape`.

However, this syntax isn't very appealing, and makes working with these new types kind of tedious. Here is a better way of doing it:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     }
```

In this case, we've defined a new type called `Person` (whose type constructor is also called `Person`), and given it named fields of specific types. Now, if we define a person, we can use automatically-generated functions to access those fields:

```
> let p = Person "Dylan" "Allbee" 100 150 "867-5309" "Vanilla, duh"
> flavor p
"Vanilla, duh"
```

There we go! That's a much nicer way to work with the constructed variable.

Now, this may seem crazy, but types can actually have parameters. Here's a classic example from Haskell itself:

```
data Maybe a = Nothing | Just a
```

This is the Maybe typeclass, which is really just a type with a variable in it. Let's say you want to write a function which will either return a value or nothing at all, you use `Maybe`:

```
maybe_get :: [Int] -> Maybe Int
maybe_get []     = Nothing
maybe_get (x:xs) = Just x
```

This function will either return the first element of the list, or nothing. And see that `Maybe Int` there? That's a concrete version of `Maybe a`, where `a` is `Int`. In this case it means that the `Maybe` type has an `Int` internally.

You can also pattern match against `Maybe` (and any other algebraic data type):

```
num_or_zero :: Maybe Int -> Int
num_or_zero Nothing = 0
num_or_zero Just x  = x
```

This function either extracts the number from `Maybe`, or it returns `0`. That's what it looks like when you pattern match against an abstract type.

Interestingly, `[]` (the list constructor) is itself an abstract type. Another way of writing the type is `[] a`, although it's usually written `[a]`.

Input and Output

Input and output in Haskell are done through something called Monads. Essentially, Monads in Haskell are a typeclass fulfilling certain laws, called the Monad Laws (which you do not need to know for now). Here is what the definition of a Monad looks like in Haskell.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n
```

Essentially, Monad is a typeclass defining four functions (it actually defines five, but these are the only four we care about). Monads are essentially a "computational context", but let's investigate what that means.

The `return` function given above is not like `return` in other languages you're familiar with. It has nothing to do with returning a value from a function, instead, it takes a value of some type a and lifts it up into the context `m`.

The next function `(>>=)` takes in something in a context, and a function that goes from a to some b also in the context, and in the end it all returns something type b in the context.

`(>>)` is similar to `(>>=)`, but it doesn't take in a function, instead taking in a value directly. Essentially, it's used for chaining operations together within the context, while `(>>=)` is used for transformations.

The final one `m >> n` is just `(>>)`'s default implementation when used as an infix operator. You can see it's just a transformation `(>>=)` that doesn't actually transform anything, meaning it just chains operations together (just as I said).

What does this have to do with IO? Well, IO in Haskell is done with Monads, like this:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

In this case, `putStrLn` is a function with the type signature `String -> IO ()`, meaning it takes a string and outputs it in the IO context.

Things are done this way to avoid the combination of "pure" and "impure" logic. Remember that "pure" means that the function should always do the same thing for a given input. But IO inherently allows for things to be unpure, because the result may change based on some input or output value. By making IO into a Monad, which is represented in the type signature, the compiler forcefully separates it from the other "pure" logic.

Now, for longer functions, Haskell actually provides a nice notation for working with IO:

```
main :: IO ()
main = do
    a <- getLine
    putStrLn a
```

This is “do notation”, and it’s used all over the place in Haskell programs to handle IO in a nice clean way. It actually looks (and usually works) fairly similarly to the structured programming style you’re actually used to.

Let’s look at what’s happening in this specific example. The `a <- getLine` works like this: `getLine` is a function with the type signature `getLine :: IO String`. The <– operator is called “bind”, and it pulls the `String` out of the IO Monad, making a just a normal `String`. Then `putStrLn` takes in a `String`, and outputs it to the console, returning with type `IO ()`.

These are the basics of IO in Haskell. When working in the IO Monad, the most important thing to remember is whether something is currently wrapped up in the IO context, or if its a plain old value.

An Actual Program

You’ve learned a lot thus far, and you may not understand it all, but now it’s time to make an actual Haskell program.

The program we’re going to make it one that takes in a credit card number (don’t worry, I have fake numbers you can use to test it), and checks whether that number is valid or not, and tells the user.

Before we start, it’s important to describe how credit card numbers are verified. The process works like this:

- Double every second digit, start from the right. So the last digit is unchanged, and the second from the last digit is doubled. Repeat all the way to the left. So, `[1,3,8,6]` becomes `[2,3,16,6]`.
- Split the new numbers into a new list of digits, and sum them up. So `[2,3,16,6]` is `2+3+1+6+6 = 18`.
- Take that number modulo 10, and check whether it’s equal to 0. If it is, the number is valid. If it’s not, the number is invalid.

Before we work on the IO, let’s get the actual credit checking logic and make sure that works.

The first thing to do is write a function that takes in a number and splits it into digits. This will actually be easier to do in reverse, so let’s write two functions. The type annotations would look something like this:

```
toDigits :: (Integral a) => a -> [a]
toDigitsRev :: (Integral a) => a -> [a]
```

So both of these functions take in some sort of `Integral a`, and return a list of the same type. Makes sense. Next we’ll want a function to do the digit doubling. The type annotation should look like this:

```
doubleEveryOther :: (Num a) => [a] -> [a]
```

This makes sense too. It takes in a list of numbers (we’re making it a bit broader because we don’t absolutely need an integer type here, but we do in the previous functions), and returns a list of numbers.

Then we want a function to sum up all the digits. That signature looks like:

```
sumDigits :: (Integral a) => [a] -> a
```

Once again, it makes sense. It takes in a list of `Integral` numbers, and returns a single number. Finally, we want a function to do the entire validation, which will look like:

```
type CreditCardNumber = Integer
data Validity = Invalid | Valid

validate :: CreditCardNumber -> Validity
```

We don't technically need to define our own type here, but it's nice to. This function takes in a credit card number, and returns whether it's `Valid` or `Invalid`. Then we can pattern match on the answer to selectively do our output at the end.

Now, let's get to writing these functions. First up, `toDigits` and `toDigitsRev`:

```
toDigits :: (Integral a) => a -> [a]
toDigits n = reverse (toDigitsRev n)
```

`toDigits` is just the verse of `toDigitsRev`.

```
toDigitsRev :: (Integral a) => a -> [a]
toDigitsRev 0 = [0]
toDigitsRev x = x `mod` 10 : toDigitsRev (x `div` 10)
```

This one pattern matches on the input. If it's 0, just return a list of nothing but 0 itself. Otherwise, take the number mod 10, and make that the first item in the list, then pass the number divided by 10 recursively to `toDigitsRev`. Basically, this goes digit by digit starting in the 1's place and moving left, constructing the list as it goes. This is why I said it would be easier to write `toDigitsRev`, and why `toDigits` is as simple as it is. With this, we can now split up numbers into digits.

Next, we need to implement `doubleEveryOther`:

```
doubleEveryOther :: (Num a) => [a] -> [a]
doubleEveryOther n =
    let pattern = cycle [1, 2]
    in reverse (zipWith (*) pattern (reverse n))
```

This is a little ugly (two reversals), but it works. Basically, it first constructs an infinite list of `[1,2]` repeated forever (that's what `cycle` does). Then it reverses the input list, and multiplies the elements from pattern and that reversed list one by one (that's what `zipWith` does). Then it reverses the final list. This is how we get the "starting from the right" behavior described in the algorithm.

Next we need to sum up the digits like this:

```
sumDigits :: (Integral a) => [a] -> a
sumDigits n = sum (concatMap toDigits n)
```

`concatMap` is new. Basically, it takes a function that generates a list, and applies it over an existing list, flattening as it goes. Without the flattening, `map toDigits n` with n as `[15,5]` would give us `[[1,5],5]`, which is not what we want. Instead, `concatMap` gives as `[1,5,5]`, which *is* what we want.

Finally, there's the validation function, which brings it all together:

```
type CreditCardNumber = Integer
data Validity = Invalid | Valid

validate :: CreditCardNumber -> Validity
validate n
    | check == 0 = Valid
    | check /= 0 = Invalid
    where check = (sumDigits (doubleEveryOther (toDigits n))) `mod` 10
```

This does exactly what you would expect. It converts the number into digits, doubles every other digits, sums them up, and then takes that whole thing mod 10. Finally, it checks if the result is 0, and returns the correct Validity value. Congratulations, you can now check credit cards!

All that's left then is to add in the IO logic, which should look something like this:

```
result :: Validity -> IO ()
result Valid   = putStrLn "Hooray! Your number is valid!"
result Invalid = putStrLn "Sorry, your number is invalid."

main :: IO ()
main = do
    putStrLn "Input a credit card number:"
    number_str <- getLine
    let number = read number_str :: Integer
    result (validate number)
```

Put all of that together, and you have:

```
import Data.List (reverse, concatMap)

toDigits :: (Integral a) => a -> [a]
toDigits n = reverse (toDigitsRev n)

toDigitsRev :: (Integral a) => a -> [a]
toDigitsRev 0 = [0]
toDigitsRev x = x `mod` 10 : toDigitsRev (x `div` 10)

doubleEveryOther :: (Num a) => [a] -> [a]
```

```haskell
doubleEveryOther n =
    let pattern = cycle [1, 2]
    in reverse (zipWith (*) pattern (reverse n))

sumDigits :: (Integral a) => [a] -> a
sumDigits n = sum (concatMap toDigits n)

type CreditCardNumber = Integer
data Validity = Invalid | Valid

validate :: CreditCardNumber -> Validity
validate n
    | check == 0 = Valid
    | check /= 0 = Invalid
    where check = (sumDigits (doubleEveryOther (toDigits n))) `mod` 10

result :: Validity -> IO ()
result Valid   = putStrLn "Hooray! Your number is valid!"
result Invalid = putStrLn "Sorry, your number is invalid."

main :: IO ()
main = do
    putStrLn "Input a credit card number:"
    number_str <- getLine
    let number = read number_str :: Integer
    result (validate number)
```

Which is just everything from before put together (with a little import at the top so we can use the `reverse` and `concatMap` functions). Let's compile it with GHC and try it out!

```
ghc --make <file name>
```

And then try it out with these two numbers: 4012888888881881 and 4012888888881882

The first one should succeed, the second one should fail. If they do, then congratulations! You have successfully made your first real Haskell program

Conclusion

I hope you've learned a lot during this workshop, and that you have some appreciation for how cool and different functional programming is. I know it may not be the easiest thing to try once you've done object oriented or structured programming for a while, but it is still worthwhile to learn and still my favorite way of writing programs.

If you want to learn more Haskell, I suggest you check out bitemyapp's "Learn Haskell" repository on GitHub, found at https://github.com/bitemyapp/learnhaskell.