Haskell Tutorial

Samuel Schlesinger

June 13, 2016

1 Purpose

Many tutorials on various programming languages are either directed towards people with little to no experience in programming, or towards people with experience in some other particular programming language. The former approach is generally structured pedantically and cautiously, as to not scare away the uninitiated. The latter approach often dives in and shows the reader code snippets side by side in order to let your knowledge in one language prop you up in your discovery of the other. I would like to accomplish neither of these, as each have been accomplished quite fully for Haskell in particular.

My focus is to show a language I love, **Haskell** to a demographic which I am a part of: **Undergraduate CS Students**, or people with a comparable knowledge base. Particularly, I will describe concepts from **Data Structures** and **Algorithms**, though I will try to include a brief explanation of each concept discussed. I also assume that you are familiar with at least one of the most popular programming languages as a part of the assumption that you have knowledge base similar to an undergraduate.

To tell you a little bit about myself, I'm a student at Clark University entering my Junior year. I study CS and maths, and currently I'm working my first internship at Autodesk. My experience in Haskell is self taught, though I want to emphasize that this is only possible through the great community that Haskell has to offer. Throughout this reading, if you have any questions, I am reachable via reddit at /u/SSchlesinger and the subreddit /r/Haskell is also a great resource, along with StackOverflow and the other usual things. If you have code that you think is interesting or cool that you've written, posting it on the Haskell subreddit is very likely to get you feedback, often even from some of the more prominent members of the Haskell community. I will likely gather a set of resources alongside this document to supplement it. This document will be hosted at Haskell Tutorial. It will be subject to changes, so if you want a particular version, make sure you save it.

2 Setup

In general, the easiest way of getting Haskell up and running is to install the **Haskell Platform**. There is support for the **Linux**, **OS X**, and **Windows** operating systems. A link which describes how to download it for each operating system can be found at `https://www.haskell.org/downloads#platform`. That being said, I currently am running the latest upload of GHC, as some nice new language extensions were included in it, and this can be found at `https://www.haskell.org/ghc/`. All code in this document should be executable from most Haskell distributions which implement Haskell 2010, the latest revision of the language. If any of it is not, please let me know.

2.1 Windows

The Haskell Platform can be installed at `https://www.haskell.org/platform/windows.html`. There are additional instructions for setting up, but recently when I installed it on my work machine, WinGHCi didn't run perfectly for me, so I just added the GHCi executable to my path and ran it from cmd.exe.

2.2 OS X

There are a variety of ways of installing for OS X which can be found at `https://www.haskell.org/platform/mac.html`, and I won't pretend to know how any of them work because I've never used a Mac for this sort of thing. That being said, I know most people use brew, and I found a source saying that the following command should get it:

```
brew cask install haskell-platform.
```

2.3 Linux

The Linux installation can be found at `https://www.haskell.org/platform/linux.html`. I use Ubuntu, and the easiest way to get it should be:

```
sudo apt-get install haskell-platform
```

2.4 Environment

For your environment, I have no IDE to suggest. The best resource out there to replace code completion is Hoogle, a search engine through many Haskell libraries, including the standard ones. I will assume you will be using the terminal and that you have the commands **ghc** as well as **ghci** on your path going forward. If this is not clear or these are not working, please feel free to contact me for help.

3 First Principles

I said that I wouldn't be starting from the basics and I won't, but even if you know how to program, if you haven't used Haskell I would read through this section. The concepts in here will not just follow you through the rest of this tutorial, but they will hopefully carry you.

3.1 Definitions

In Java, C, Python, as well as the majority of modern programming languages, assignment is an action. This action is denoted by a variable name on the left, an equals sign, and some sort of value on the right. This notation is used in Haskell for something much stronger, which is something I'll refer to lightly as equality, but could be more strongly called propositional equality. This means that when I define something to be equal to something else, I can interchange these two things anywhere this definition is in scope and I will not modify the meaning of whatever greater expression they are a part of. As we all know, assignment is useful for a variety of reasons, not the least of which are brevity and control. This stronger version of assignment is useful for the same reasons, but it also gives us something called **referential transparency**, the strong notion of equality I described above, which lets you reason about the values you're representing without even discussing the memory locations you have to consider while using other programming languages. This is something typical of what is called the functional programming paradigm.

```haskell
myHugeNumber = 10000000000000000000000000000000000000000000

myNewString = "yada yada yada"

sliceOfPi = 3.14
```

The definition I gave above might scare some of you in the following way. If you were writing some code in an imperative programming language like C#, you might have defined a class which contains some variable you'd like to play with. As such, you might define the set and get methods for that variable and then change the value as you see fit. This immediately contradicts the notion I defined above. If you can change the value of something you've previously defined, then you can't simply replace the right and left hand sides of your definitions with each other willy nilly. As such, definitions are final in Haskell. They are similar to mathematical definitions in this way. Though this may seem weird to some, it allows for much nicer analysis of your programs in many cases. Above, myHugeNumber literally refers to that massive number on the right. It doesn't just have it currently stored in whatever location the symbol myHugeNumber represents, it actually **is** equivalent to that huge number, which is a much stronger statement than you'll get in almost any other programming language.

There are two fears that I believe come up when one learns this fact about Haskell and its immutable definitions. The first I see is about efficiency, asking how on earth one can make efficient algorithms if one has to create new variables whenever they want to change something. This is a valid concern, but the important thing to realize is that this strong notion of equality is true when it comes to what your code **means**, not necessarily in terms of how the compiler will implement it. The next concern has to do with certain patterns that people have learned to rely on quite heavily in the average CS curriculum, especially looping constructs (hopefully not gotos). I can assure you that there are replacements for these which, if written in the proper way, will compile to essentially the same machine code as your for loops do.

3.2 Names

Each name in scope distinctly refers to one value. Type names must begin with a capital letter whereas variable names must be lowercase.

3.3 First Types

No matter what modern language you use, there is some notion of a type. The general idea of a type is that it is some meaningful annotation alongside your values and variables which indicates what sort of thing each one is or is allowed to be. In C, this is a very weak notion, as you are allowed to cast types back and forth from one another and really the only hard and fast type you get is a strip of bits of some length hiding away in your computer. In Java, there are atomic types such as int and char, and then there are types which extend from Object, the base for creating data structures in the language. You are allowed to a large extent to change things from one type to another as you see fit.

In Haskell, the notion of a type has a very strong will. There is no automatic coercion between types, and to take a type to another type without a proper transformation you have to use an unsafe function that I will not name here. That being said, certain literals are overloaded such that the compiler will deduce what they are supposed to be and they will become that type. In particular, numbers will do this, as you can check for yourself by entering the following code into GHCi and using the :t ¡thing¿ command to inspect the types. The code below also shows the notation for asserting that a given name will refer to something of a certain type, as well as illustrates that the Integer type and the Int type are distinct, the first referring to the mathematical integers (within the bounds of your memory) and the second to a 32 bit Int (though technically I think the specification says that it only has to be 30 bits).

```
myFloat :: Float

myFloat = 10
```

```haskell
myInteger :: Integer

myInteger = 10412421512512

myTuple :: (Integer, Float)

myTuple = (100, 53)

myList :: [Char]

myList = "hello yes this is a string"

mrTrue :: Bool

mrTrue = True
```

3.4 Function Types

We've seen a few basic types, but so far we've left out something rather important for programming, especially in a functional language, and that's functions. In Haskell, functions are supposed to mirror what they are defined to be in mathematics, and that is that every time a function f takes some input x, f(x) will return the same value. That being said, in Haskell we usually write f x instead, as we use so many functions that having all of those parentheses hanging around can distract from what's really going on. Concise and easy to read code is definitely a virtue.

```haskell
isZero :: Integer -> Bool

isZero n = case n of
  0 -> True
  _ -> False
```

The above code gives us our first example of a function along with pattern matching. All the case construct does is take its argument n and try to match it with each pattern below in order. If it matches up with one, it evaluates the expression to its right. There is a more concise notation for this which is equivalent, shown below.

```haskell
isZero’ 0 = True

isZero’ _ = False
```

There is another way we could have written this code, using something called a guard, which is similar to pattern matching but uses expressions of type Bool to determine what to evaluate to.

```haskell
isZero'' n
  | n == 0 = True
  | otherwise = False
```

Don't let otherwise fool you, it isn't anything fancy, it's simply defined as otherwise = True in the Prelude, which is the standard bunch of definitions which are automatically imported whenever you start up GHCi or compile your code with GHC. I have one more version of this function for you before we move on...

```haskell
isZero''' n = n == 0
```

3.5 User Defined Types

3.5.1 Type Synonyms

There are a three ways that I know of to define types in Haskell. The first is the simplest one and it's solely present for the programmers pleasure, and this is a **type synonym**.

```haskell
type Vector3D = (Float, Float, Float)
```

Cool, we've defined a Vector3D! Now whenever I want to write functions to do my physics homework, I can just write functions that give and take Vector3D's instead of (Float, Float, Float)'s. This is honestly more convenient than you'd think, and if used properly can really serve to let your code document itself. It has zero runtime overhead and it allows you to see the see what your code means later on if you use it right. Personally, I often use these in such a way that when I describe my algorithm out loud, I get to use the same vocabulary as within the code itself. That way, if I can get someone to understand a program with language, there is a mostly one to one mapping to the code from this initial comprehension.

3.5.2 Data Declarations

Though these type synonym guys are cool, they don't really add anything to the representational capacity of the language. In order to add types which are actually new sorts of data we can encounter in our computation, we use something called a **data declaration**. This lets us define our own types of things in a very flexible way, but in a way that might seem too simple or "dumb" (in the sense of a dumb data type) , particularly coming from the land of object orientation. As an example, we could have defined our Vector3D type in the way we're about to define a Vector2D.

```haskell
data Vector2D = Vector2D Float Float
```

To break down the above statement, it says that a Vector2D is a type, and the only way to construct it is to use the **data constructor** Vector2D, which, if we were to inspect in GHCi, has type Float -¿ Float -¿ Vector2D. Now we can use this with pattern matching in order to do define some operations for ourselves.

```
addVector2D :: Vector2D -> Vector2D -> Vector2D
addVector2D v w = case v of
  Vector2D a b -> case w of
    Vector2D c d -> Vector2D (a + c) (b + d)
```

Great! This is very similar to what we did above. We can simplify this in the same way that we could for isZero'.

```
addVector2D' (Vector2D a b) (Vector2D c d) = Vector2D (a + c) (b + d)
```

We could also have written our Vector2D in a slightly different way, using something called **record syntax**.

```
data Vector2D' = Vector2D' { x :: Float, y :: Float }

myGoodVector = Vector2D' { x = 421, y = 0.21 }

xOfMyGoodVector = x myGoodVector
```

The above is a syntactic sugar to simulate the object/struct paradigm which can be found in C and Java. One particularly good use case for it is when you have a data type which you'll be adding or removing fields to, perhaps because you haven't decided what you need to store in it, but even if you use this notation, you can still pattern match on Vector2D' f1 f2 and use it as a normal constructor.

Something else we can do in a data declaration is declare a bunch of different sorts of things a type might be. This could be useful if we're defining a common type which can take many different forms. The most common way in which this is used is if we have some function which might return some value that we want, but it also could not have anything for us.

```
data PerhapsInt = GotOneHere Int | NotToday
```

This time, we had to name our data constructors something different than our type name, cause there's more than one of them. That being said, we've already seen a type like this. A list either has an element and another list, the tail, or it can be empty. The constructors for lists are (:) and [], and I'll use them below to make this apparent.

```haskell
type IntPairs = [(Int, Int)]

lookupTheInt :: IntPairs -> Int -> Int
lookupTheInt ((x, y):rest) k
  | k == x = y
  | otherwise = lookupTheInt rest k
```

This function does a rather obvious thing. Given a list of IntPairs, it checks if the first pair's left element matches the key we're searching for. If it does, we return the value on the right, otherwise we lookupTheInt in the rest of the list. What happens if we reach the empty list, though? Try it in GHCi. What you should see is an Exception with a message saying that there are ”Non-exhaustive patterns in lookupTheInt”. If you look at lookupTheInt's definition, this is apparently true. There is no match for lookupTheInt [] k, so when the code compiles it puts an error there. To be explicit, the pattern matching above will be converted into a pattern matching case statement, and in the bottom, there will be a _, which matches anything, and it will probably return an error with an error message similar to what you saw. In order to avoid this problem, we use PerhapsInt.

```haskell
maybeLookupTheInt :: IntPairs -> Int -> PerhapsInt
maybeLookupTheInt [] _ = NotToday
maybeLookupTheInt ((x, y):rest) k
  | k == x = GotOneHere y
  | otherwise = maybeLookupTheInt rest k
```

In fact, this pattern is so common in Haskell that, in the Prelude, there is a type which does exactly what our PerhapsInt does, but for any type. It's called Maybe. We'll get to that very shortly.

3.5.3 Newtype

This is the last sort of type definition, and it is a bit less intuitive as to why you might want it. Basically, it's the same as a data declaration, except it's called **newtype** instead, and it can only contain one data constructor and one field.

```haskell
newtype HiddenInts = HiddenInts [Int]
```

There are various reasons one might want to use a newtype declaration, but the most obvious is that, unlike a type synonym, it lets you control what people get to do with your new type. For instance, maybe you don't want to show people what's inside your list and you export that without the internal definition. Then, you can export whatever functions you want people to be able to act on it with and you have much more fine grained control over your data structure's invariants, even if internally all you use is something everyone can just pattern match over and take stuff out of.

3.6 Polymorphism and Type Classes

3.6.1 Polymorphism

There are some functions and data structures which don't just work for Int, Float, or Bool, but for everything in the same way. In a language like Java or C++, polymorphism is ad hoc, which means that in any given place, one has to instantiate your polymorphic type with a type, like Collection¡BlueBird¿. In Haskell, the polymorphism is parametric, which means that you can use type variables in your function and data declarations. I'll display this below.

```
theSame :: a -> a
theSame x = x

data Perhaps a = Indeed a | Alas
```

Now we have a function theSame which takes something of any type a and returns whatever x you give it, and a type Perhaps a which is either Indeed an a or Alas, not an a. In the Prelude, theSame is called **id** and Perhaps is called **Maybe**, with data constructors **Just** and **Nothing**.

3.6.2 Typeclasses

In object oriented programming, a class is basically a type with functions baked into it. An abstract class, or an interface, is a description of the sorts of functions you must bake into some type that inherits from it, extends it, what have you. A class in Haskell, referred to as a typeclass, is something more like an interface than anything else, but is an extension of this idea. We could take things from there, extending the concepts and keeping our past knowledge, but this is hardly creative. Let's try to discover some places where we might need them, given all of the tools we already have.

```
containsAttempt :: [a] -> a -> Bool
containsAttempt = undefined
```

If we were to attempt to write this function, we'd immediately run into a problem... How do we check whether or not the a in our list is equal to the a we were given? We could use the == function we've been using in our earlier functions, but it turns out that actually is a function that belongs to a class called **Eq**. The way we can demand that our contains function only takes types that are an **instance** of Eq is shown below.

```
contains :: (Eq a) => [a] -> a -> Bool
contains [] _ = False -- No a is in the empty list
contains (x:as) a
  | a == x = True
  | otherwise = contains as a
```

Lets try to solve a harder problem, the problem of sorting a list. We need to have some notion of order to do this, and it turns out the concept of a total ordering is described in the class **Ord**.

```
sortList :: (Ord o) => [o] -> [o]
sortList [] = []
sortList (x:xs) = sortList [y | y <- xs, y < x] ++ (x : sortList [z | z <- xs, z <= x])
```

This introduced some new notation, just because I think this is so beautiful, but it's pretty intuitive. Basically, this is stating that the empty list is already sorted, and that if we have some element at the front of our list, we can recursively sort that list by taking all the elements that are less than the front element, sorting and putting them to the left, the front element in the middle, and all the elements greater than or equal to the front element to the right of it after sorting them. This is derived from a similar line of thought as quicksort, but doesn't have the same performance characteristics. In order to implement it for arbitrary lists, we need the **Constraint** Ord o, which is a sort of predicate that demands there must be a function for comparing things of type o.

That brings us a little closer to our definition, the idea that a class is a sort of requirement for a type, or a constraint. That's basically what we want when we have these sorts of desires for types which have certain qualities, a type which has a certain group of functions defined upon it in some way or another. This is similar to an interface, however an interface, at least in the languages which I've used them in, has the limitation that the first argument of each function specified must be the type which instantiated it. A class in Haskell, on the other hand, can even request that there exists a function that returns the type instantiating it. I'll now define our first class, and define one instance for it.

```
class Testable t where
  test :: t -> Bool

instance Testable Bool where
  test b = b
```

The class declaration says that if t is Testable, there exists a function test which maps t to a Bool, presumably to True if the test was passed and to False if the test failed. The instance declaration says hey, if I have a Bool already, then testing it should be as simple as returning it. So for Bool, test is equivalent to id. We'll come back to this class later.

4 Exploration: Chains

Lets use the tools we've just discussed and explore what we can do with a simple structure, something I'll call a Chain.

```haskell
data Chain a = Link a (Chain a)
```

So now, whenever we have a Link a, we can pull out the first a, and we can get the rest of the chain. We can also imagine this function as one thing, similar to popping from a stack, where we get out the first element and the rest of the Chain.

```haskell
first :: Chain a -> a
first (Link a _) = a

rest :: Chain a -> Chain a
rest (Link _ cs) = cs

pop :: Chain a -> (a, Chain a)
pop (Link a cs) = (a, cs)
```

Cool, so now we have ways of taking things out of Chains, but how can we construct them? Well, in most programming languages, arguments to functions are evaluated strictly, so if some function call goes into an infinite loop, then your whole program halts. This is not the case with Haskell. In Haskell, items are evaluated lazily, which means that, until the result of some diverging computation is needed, it won't cause the program to halt. This lets us construct infinite chains via recursion, as below.

```haskell
forever :: a -> Chain a
forever a = Link a (forever a)
```

This is great, but all it really gives us is a way to create infinite constant sequences... Let's see what else we can do. Lets try to make some functions which complement the ones we already have. Particularly, we can look at pop, which takes a Chain and splits it into the rest and the first. Lets try to come up with the inverse of this function, which we can call push.

To do so, let's set some rules: push (pop c) = c and let's do some reasoning.

```haskell
push (pop (Link a cs)) = push (a, cs) = Link a cs
```

From setting this identity and reasoning from there, we've come up with a definition for push which merely involves the parameters it needs to take. We can now define this.

```haskell
push (a, cs) = Link a cs
```

What other functions should we try for? Well, what about a function which takes another function and maps every single element of our chain through it? In other words, if we have a Chain of all of the positive numbers, and we want to get all the even numbers, we can just pass our chain and the function that multiplies things by two through this mapping function. The axioms we'd want to satisfy for this function, which we'll call fmap, would probably be as follows:

```
fmap id x = x

fmap f (fmap g x) = fmap (f . g) x
```

That is to say, if we map the identity function over our chains, we want them to stay the same, and if we map g and then map f over a chain, it should just be the same as doing g composed with f the first time. It turns out there is a typeclass for things of this exact form called Functor. It turns out that, if we want our fmap to typecheck, we can really only define it in one way.

```
instance Functor Chain where
  fmap f (Link a cs) = Link (f a) (fmap f cs)
```

The reason that this is the case is simple. It's the only version of this function that will obey the type system's demands. Let's break this down a bit further by looking at the types of each item in the above code.

```
fmap :: (a -> b) -> Chain a -> Chain b

f :: a -> b

Link :: a -> Chain a -> Chain a

a :: a

cs :: Chain a

-- So our goal is, given a (a -> b) and a Chain a, to produce a Chain b.
-- The only constructor we have for Chain a is Link a (Chain a), so we
-- need to use our available functions and values to transform that into a Chain b.

fmap f (Link a cs) = ? -- Well, first we need to deal with that a to make it a b.

fmap f (Link a cs) = Link (f a) ? -- Well, we have a Chain a, cs, and we have fmap,
-- which, given a function (a -> b), will transform our Chain a into a Chain b.

fmap f (Link a cs) = Link (f a) (fmap f cs) -- We're finished!
```

Scary Proof Part Before we really can say we're finished, we should show that this function obeys the rules we set up for it. To show that it obeys the axioms, we can use a method called induction. Induction is a method of proof which works very well for data types defined like those in Haskell, but we'll talk about it specifically in this case. If we want to show that one Chain is equal to another, we need to show that each element in one is equal to the corresponding element in the other. Our inductive argument will go like this for each axiom: If the chains are equal up to the first element, and if the chains being equal up to the N-th element implies that they are equal up to the N+1-th element, then the chains must be equal.

```
-- First axiom: fmap id x = x
-- Base case: Equal up to the first element
fmap id (Link a cs) = Link (id a) (fmap id cs) = Link a (fmap f cs)

-- Now let's assume that the chains are equal up to the N-th element
Link a1 (Link a2 ... (Link aN cs)) = Link a1 (Link a2 ... (Link aN (fmap id cs)))

-- Then we can prove that they are equal up until the N+1th
Link a1 (Link a2 ... (Link aN (Link aN1 cs)))
= Link a1 (Link a2 ... (Link aN (Link aN1 (fmap id cs))))

-- Thus, all of the elements are equal.

-- Second axiom: fmap f (fmap g x) = fmap (f . g) x = x
-- Base case: Equal up to the first element
fmap (f . g) (Link a cs) = Link (f (g a)) (fmap (f . g) cs)

fmap f (fmap g (Link a cs)) = fmap f (Link (g a) (fmap g cs))

fmap f (Link (g a) (fmap g cs)) = Link (f (g a)) (fmap f (fmap g cs))

-- Again, let's assume that the N-th elements are equal.
Link b1 (Link b2 ... (Link bN (fmap (f . g) cs) = Link b1 (Link b2 ... (Link bN (fmap f (fma

-- Then we can again show that they are equal up until the N+1th element.
Link b1 (Link b2 ... (Link bN (Link (f (g aN1)) (fmap (f . g) cs))))
= Link b1 (Link b2 ... (Link bN (Link (f (g aN1)) (fmap f (fmap g cs)))))
```

Our implementation checks out! This might seem counterintuitive, but many classes are similar to Functor in that they demand the implementer obey some axioms in their instances. That being said, we won't go through the proofs every time, particularly for something like this where our implementation obeying the axioms was actually quite obvious.

5 Data Structures

I know some people probably got a little fearful of the last section, but don't fret if you didn't quite understand it yet. Right now we're going to cover some much simpler stuff that should feel very familiar. Let's implement all of our favorite data structures!

5.1 Linked Lists

Let's start with the basics and code up a linked list! For those of you who don't know, a linked list is a way of storing lists where you keep a whole bunch of links, each of whom store a piece of data and knows where the next guy in the list is. A linked list can also be empty. We've already seen this in Haskell with the built in list type, but let's implement our own version to get the hang out things a little bit, just by modifying the Chain type we already made.

```
data List a = Link a (List a) | Empty
```

This list is great, but it is quite a dumb data structure. Let's define some operations on lists... I'm going to start reusing some names that are defined in the Prelude, so if you want to define some of these things on your own and start playing with them, you'll have to change the names. I personally always change the names of functions in code that I'm learning from, it helps me understand it without being able to refer back to the file whenever I get confused about the role of something.

```
length :: List a -> Int
length Empty = 0
length (Link a rest) = 1 + length rest

repeat :: a -> Int -> List a
repeat a n | n < 1 = Empty | otherwise = List a (repeat a (n - 1))

get'kth :: List a -> Int -> Maybe a

get'kth Empty _ = Nothing
get'kth (List a rest) k | k < 0 = Nothing | k == 0 = a | otherwise = get'kth rest (k - 1)

append :: List a -> List a -> List a
append Empty stuff = stuff
append (List a rest) stuff = List a (append rest stuff)
```

You should be able to understand these functions at this point if I've gotten my messages across properly, but I want to explain how each of them would execute (if they were needed, recall laziness) in order to really drill the point home.

length This is probably the simplest one to understand and it is basically a definition. The length of an empty list is 0. That makes sense I hope. Then, in order to find the length of a list with some element and the rest of the list hanging off of it, we add one to the length of the rest of the list. If at this point this doesn't make sense please contact me on reddit and I will try to explain better.

repeat This one is actually quite similar to length, but we get the length as an argument and an element to fill a list of that length with. If the length we get is less than one, then we should just return the empty list. This is an implementation choice, so if you'd rather this function return a Maybe (List a) if you pass it a number less than 0, then be my guest and implement that (it ends up giving you a nice property involving length and repeat you could prove if you do it this way). If n is greater than or equal to one, then we simply add one a to the front of our list and make the rest of the list by repeating a n-1 times.

get'kth This function is also similar to the first two, but it simply runs through to the kth element and gives it back. This returns a Maybe, as if we get passed Empty or if k ¡ 0, then I personally don't know what a to return. If this is not the case, we check if k == 0, and return the element we're looking at if this is the case. Otherwise, we get the (k-1)th element in the rest of the list.

append This function is definitional as well. Appending a list to another means taking the first and plopping the second on its tail. If the first is empty, this just means returning the second. If it isn't empty, then it should stick the first element on the front and make the rest the rest of the list appended to the second.

5.2 Queue

Another important structure programmers often use is a queue, but this often feels clumsy to implement in a functional language, as in am imperative language we might use a doubly linked list or an array or something. We could take these approaches in Haskell, but there is an even better approach that gives us (amortized) constant time enqueues and dequeues. For reference, a queue is a structure which gives you first in, first out access, meaning that the order in which you put elements in with enqueue will be the order you take them out with dequeue. This is supposed to model in a line, or a queue.

```
data Queue a = Queue [a] [a]

enqueue :: a -> Queue a -> Queue a

dequeue :: Queue a -> (Maybe a, Queue a)
```

Basically, a Queue will be represented by two lists. The left list will represent the front of the queue, and the right will represent the back of the queue, the first element of the left being the first element of the front and the first of the right being the first of the back. So, in order to enqueue, we will need to push the elements into the front of the right list, and to dequeue, we will pull elements from the left. If we try to pull from a queue where the front is currently empty, we should flip the back and put it on the front.

```
enqueue a (Queue front back) = Queue front (a:back)

dequeue (Queue [] []) = (Nothing, Queue [] [])
dequeue (Queue [] back) = let (a:front) = reverse back in (Just a, Queue front [])
dequeue (Queue (a:front) back) = (Just a, Queue front back)
```