5 - Datatypes and higher-order functions

"After closer investigation, it's become clear that we need to enter more than one value."

Agenda & recap

Last week, we:

- Studied the boot sequence of a computer.
- Saw what an OS is.
- Used a Linux virtual machine.

Today, we’re going to write a lot of code:

- Higher-order functions: functions that transform functions!
- Datatypes
- Generalizing higher-order functions: the Functor typeclass

Remember long division? Me neither.

Previously, we looked at the `div` function, which divides numbers (obviously). But how does it work? Let’s code up our own `div` together as a refresher.

Higher order functions

One of the most powerful things about Haskell is you can have **functions** as **input** or as **output** of another function, called higher order functions (hof)

The first hof we're going to look at is `map`.

Let's look at its type signature by typing :t map in GHCi. What does it read? (Remember, we can forget Traversable t and just think of t as meaning "list".)

map takes in two inputs - a function, and a list. It then applies this function to every element in the list.

Now that we know the type signature and how it works, let's try and code our own map!

Let’s jump next to the filter function

filter is another important hof. Let’s see its type signature using :t filter.

This function **filters** a list according to the function (a -> Bool). This function tests each element of a list (hence the bool function). Elements that return false are removed from the list.

For example, we could filter a strings to select only the characters that are lowercase. That would use a function (String -> Bool). Ex.

```
filter (isLower) “DOEsNT MaKING vARIeD FUNCTIONS AmAZe YOU”
```

What would this return?

Now let’s code up filter in pairs!

Nothing special about lists

Haskell lets us define *functions*, which are a way of *transforming* things. But we can also define *datatypes*, which are a way of *representing* information.

For instance, we might want to represent a *person*. Let's define a datatype Person that contains information that is related to a person.

```
data Person = MakePerson String Int deriving Show
```

The String and Int here refer to the person's name and age.

deriving Show is necessary so that GHCi knows how to show you the values of the type we just invented.

Making people is easy!

In our datatype, we know what the String and Int do but what does MakePerson do?

```
data Person = MakePerson String Int deriving Show
```

Let’s ask `:t`! Write your current code and open GHCi, running `:t` on `MakePerson`. What does the type signature tell us?

We can think of `MakePerson` as a function that takes a name and an age and makes a person. But we call this a *constructor,* not a function because it is the *only* way we can construct a value of type Person!

Enter the matrix

Using the new datatype we just created (Person), add yourself to your code.

To do this, we need to use MakePerson and fill in the necessary information. So if I wanted to make myself, I would write:

MakePerson “Eric” 23

Like pattern matching, we need to introduce the arguments to MakePerson they way they are written when we defined Person. This means after using MakePerson, we have to add our name as a string then our age as an Int. Doing it in the opposite order would not compile.

Now we have this data, we need to bind it to something!
Ex: me = MakePerson “Eric” 23. Let’s compile this and ask GCHi who “me” is.

Working with people is easy!

Now let’s write a function creates a child for a person.

```
child :: Person -> Person
```

It should suffix “Junior” to the parent’s name, and the new person should have age zero. Let’s do this together.

We can pattern match on own datatypes the same way we did for earlier types like lists! To do this, we need to introduce arguments that correspond to the *constructors*.

The right-hand side is where we can build our new person, with a different name and age.

Happy birthday to you!

Let’s also make a function that makes people get older.

```
birthdayParty :: Person -> Person
```

This function takes a person, sees what their age is, and constructs a new person with the same name, but whose age is increased by one. Let’s code these together.

Higher-order people

Let’s combine our knowledge of datatypes and higher-order functions! Code this up individually:

```
rename :: (String -> String) -> Person -> Person
```

which transforms a person’s name according to the given function. Let’s make a function that adds “ the Great” to any name.

```
makeGreat :: String -> String
```

Try this out as a test:

```
rename makeGreat me
```

Parent-child bonding is essential!

Let’s represent the fact that parents and children are related, by adjusting our definition of Person:

```
data Person = MakePerson String Int Person (don’t write this yet)
```

Now each person also contains another person, which we’ll say is their parent. But what if we don’t know who someone’s parent is? Who are the first people? Very important questions.

We need a way of specifying that something is *optional* using Maybe.

Yes. No. Maybe. Can you repeat the question?

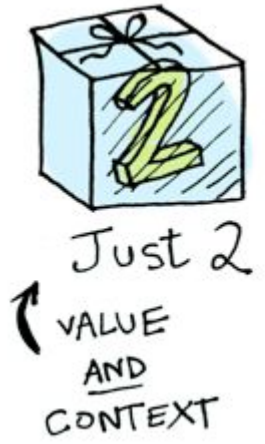
What is Maybe? Ask GHCi! Because Maybe isn't a function and instead a type (hence the uppercase M), we need to use :info instead of :t. The command looks like this:

```
:info Maybe
```

GHCi just spat out a bunch of text, but let's look at the first line:

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

The pipe | can be read as "or", meaning a (which is polymorphic) can either be Nothing or something ("Just a", in other words Just itself. This can help us prevent our code from crashing!



Maybe parents

If we have a value of type Maybe Int, then we might have an Int or we might not. This is useful for representing *optional information*.

Notice that this is a **polymorphic type**: there is a *type parameter* a that can stand for any type we want.

This type has *two constructors*, not just one!

Parents: now optional!

We can use Maybe when we add the parent field to our Person datatype. It's syntax looks like this:

```
data Person = MakePerson String Int (Maybe Person) deriving Show
```

Now each person we create *might* have a parent. Let’s adjust our all our functions from earlier that call MakePerson so it has the right number of inputs!

Let’s also make a function for constructing people with no parent:

```
person :: String -> Int -> Person
```

Family trees

Challenge: write a function that computes a person’s lineage:

```
lineage :: Person -> [String]
```

It should return the person’s name, followed by their parent’s name, and so on, until we hit the ancestor that doesn’t have a parent.

Can't spell functor without fun!

Notice how Maybe wraps around a, which can be of any type. In what way is this similar to List?

Both wrap around types that contain values of that type! These are called functors. Jake what is a functor?

Yet another map

Remember map :: (a -> b) -> [a] -> [b]? This function applies the same function (a -> b) to all values wrapped in the [] functor, but this won't work for types wrapped in the Maybe functor.

It turns out we can code a similar function for Maybe!

```
omap :: (a -> b) -> Maybe a -> Maybe b
```

Try to implement it!

One map to rule them all!

Right now we have a map for the [] functor and Maybe functor, but what about map for any kind of functor? Let’s write a *generalized* map, called fmap’!

```
fmap’ :: (a -> b) -> f a -> f b
```

Where f is any possible functor! Let’s implement this.

We did it!

Today we learned about:

- Higher order functions
- Datatypes
- And functors!

In 2 weeks from now, were jumping into web dev as we need to make those websites! So we’ll look at:

- CSS and HTML

For homework, try codacademy’s HMTL tutorial.

Functors: a generalized map

Functors are all those type constructors that support the notion of applying a function uniformly to the stuff inside.

For example, the `[]` (list) type constructor is a functor because we have the function `map :: (a -> b) -> [a] -> [b]` that turns any ordinary function into a function on lists.

Similarly, Maybe is a functor because we have `omap :: (a -> b) -> Maybe a -> Maybe b` that turns any ordinary function into a function on `Maybe`.