Types

Types are ways to categorize data/value/expression/function in haskell. In interactive mode, the command `:type` or `:t` followed by a value gives the type of the value.
`::` is read as "has type of"

Types information

Read type signatures, type declaration, function type signature

Activity

1. Open up your Haskell GHCI REPL.
2. run the command `:type 't'` and how do you interpret the result?
3. As in the previous question run the command `:type “Takam”` and interpret the result
4. The same for the command `:type True`
5. (Polymorphic) run the command `:type 13` Interpret the result. We say that 13 is a polymorphic value. To see the differents types (forms) of 13 can take, run the command `:info Num`
6. (Function type) Run the command `:type not` and how do you interpret the result? Run the command `:type (+)` and how do you interpret the result? (Operator infix and prefix)

Lesson:

- In the Haskell REPL we can have the type of a value by running the command `:t` or `:type` followed by the value.
- A value can have more than one type. We say that the value is polymorphic.
- We can have information about Typeclass by running the command `:i` or `:info` followed by the type class name.
- In the definition of function we can constrain types to belong to certain Typeclass (to implement the typeclass) that expose a number of operations. This is done by using this syntax:
  `f :: [(Constraint1,Constraint2, ...) =>] inputType -> outputType`
  Example: `f :: (Eq a, Num b) => a -> b`
  This means that the type a must implement the typeclass Eq (or belong to the typeclass of value that can be equated) and the type b must implement the typeclass Num.

List type

An intro to lists

Much like shopping lists in the real world, lists in Haskell are very useful. It's the most used data structure and it can be used in a multitude of different ways to model and solve a whole bunch of problems. Lists are SO awesome. In this section we'll look at the basics of lists, strings (which are lists) and list comprehensions. In Haskell, lists are a homogenous data structure. It stores several elements of the same type. That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

Annotation

In Haskell, lists are denoted by square brackets and the values in the lists are separated by commas.
Example:

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

If we tried a list like [1,2,'a',3,'b','c',4] , Haskell would complain that characters (which are, by the way, denoted as a character between single quotes) are not numbers.

Strings

After playing for some time with characters, you may wonder whether you can have a bunch of them together, forming what is commonly known as a string. The syntax for strings in Haskell are similar to C: you wrap letters in double quotes. The following code creates a string. If you ask the interpreter its type, what do you expect to get back?

```
Prelude Data.Char> :t "Hello world!"
"Hello world!" :: [Char]
```

Instead of some new type, like String, you see your old friend Char but wrapped in square brackets. Those brackets indicate that "Hello world!" is not a character but a list of characters.

generalization

In general, given a type T, the notation [T] refers to the type
of all lists whose elements are of that type T. Lists are the most used data structure in functional programming. The fact that a type like a list depends on other types is known as parametric polymorphism, and you will delve into the details of it in the next section.

List literals (i.e., lists whose values are explicitly set into the program code) are written with commas separating each of the elements, while wrapping everything between square brackets. As I have said, there's also special string syntax for a list of characters.

Let's look at the types of some of these literals and the functions reverse, which gives a list in reverse order, and (++), which concatenates two lists.

```
Prelude> :t [1,2,3]
[1, 2, 3] :: Num t => [t]
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> reverse [1,2,3]
[3,2,1]
Prelude> reverse "abc"
"cba"
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Some basic functions on lists

head

takes a list and returns its head. The head of a list is basically its first element.

```
ghci> head [5,4,3,2,1]
5
```

tail

takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

last

takes a list and returns its last element.

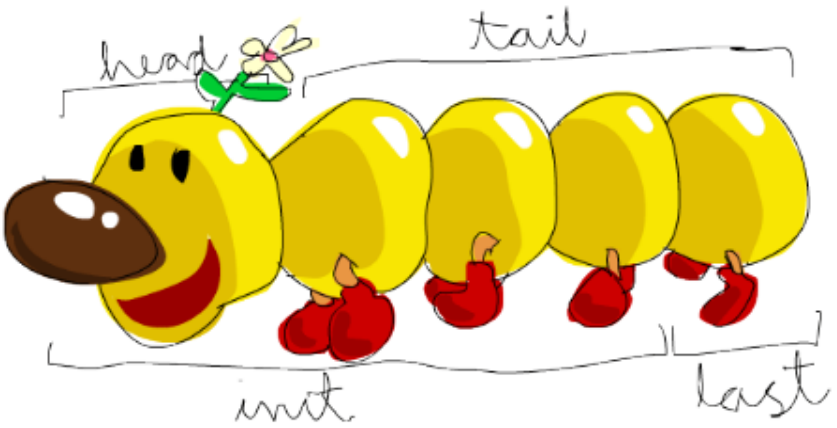
```
ghci> last [5,4,3,2,1]
1
```

init

takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

If we think of a list as a monster, here's what's what.



But what happens if we try to get the head of an empty list?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

Oh my! It all blows up in our face! If there's no monster, it doesn't have a head. When using head , tail , last and init , be careful not to use them on empty lists. This error cannot be caught at compile time so it's always good practice to take precautions against accidentally telling Haskell to give you some elements from an empty list.

length

takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

null

checks if a list is empty. If it is, it returns True , otherwise it returns False . Use this function instead of xs == [] (if you have a list called xs)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

reverse

reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

take

takes number and a list. It extracts that many elements from the beginning of the list. Watch.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

See how if we try to take more elements than there are in the list, it just returns the list. If we try
to take 0 elements, we get an empty list.

drop

works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

maximum

takes a list of stuff that can be put in some kind of order and returns the biggest element.

minimum

takes a list of stuff that can be put in some kind of order and returns the smallest element.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
```

```
9
```

sum

takes a list of numbers and returns their sum.

product

takes a list of numbers and returns their product.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

elem

takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

(!!)

takes a list of things and a position and returns the element at that position in the list. If the position is greater than the length of the list an error is thrown. It's usually called an infix function because it's easier to read that way.

```
Prelude> :t (!!)
(!!) :: [a] -> Int -> a
Prelude> [1, 2, 3, 4, 5] !! 2
3
Prelude> (!!) [1, 2, 3, 4, 5]
3
```

Those were a few basic functions that operate on lists. We'll take a look at more list functions later.

Texas range

What if we want a list of all numbers between 1 and 20? Sure, we could just type them all out but obviously that's not a solution for gentlemen who demand excellence from their programming languages. Instead, we'll use ranges. Ranges are a way of making lists that are arithmetic sequences of elements that can be enumerated. Numbers can be enumerated. One, two, three, four, etc. Characters can also be enumerated. The alphabet is an enumeration of characters from A to Z. Names can't be enumerated. What comes after“John” ? I don't know.
To make a list containing all the natural numbers from 1 to 20, you just write [1..20] . That is the equivalent of writing [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] and there's no difference between writing one or the other except that writing out long enumeration sequences manually is stupid.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Ranges are cool because you can also specify a step. What if we want all even numbers between 1and 20? Or every third number between 1 and 20?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

It's simply a matter of separating the first two elements with a comma and then specifying what the upper limit is. While pretty smart, ranges with steps aren't as smart as some people expect them to be. You can't do [1,2,4,8,16..100] and expect to get all the powers of 2. Firstly because you can only specify one step. And secondly because some sequences that aren't arithmetic are ambiguous if given only by a few of their first terms. To make a list with all the numbers from 20 to 1, you can't just do [20..1] , you have to do [20,19..1] .
Watch out when using floating point numbers in ranges! Because they are not completely precise (by definition), their use in ranges can yield some pretty funky results.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

It is not recommended to use them in list ranges. You can also use ranges to make infinite lists by just not specifying an upper limit. Later we'll go into more detail on infinite lists. For now, let's examine how you would get the first 24 multiples
of 13. Sure, you could do [13,26..24*13] . But there's a better way: take 24 [13,26..] . Because
Haskell is lazy, it won't try to evaluate the infinite list immediately because it would never

finish. It'll wait to see what you want to get out of that infinite lists. And here it sees you just want the first 24 elements and it gladly obliges.

A handful of functions that produce infinite lists:

replicate

creates a list of length given by the first argument and the items having value of the second argument

cycle

takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

repeat

takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Although it's simpler to just use the replicate function if you want some number of the same element in a list. replicate 3 10 returns [10,10,10] .

List comprehension

If you've ever taken a course in mathematics, you've probably run into set comprehensions. They're normally used for building more specific sets out of general sets. A basic comprehension for a set that contains the first ten even natural numbers is S = {2. x | x ϵN, $x \leq 10$} . The part before the pipe is called the output function, x is the variable, N is the input set and $x \leq 10$ is the predicate. That means that the set contains the doubles of all natural numbers that satisfy the predicate. If we wanted to write that in Haskell, we could do something like take 10 [2,4..] . But what if we didn't want doubles of the first 10 natural numbers but some kind of more complex function applied on them? We could use a list comprehension for that. List comprehensions are very similar to set comprehensions. We'll stick to getting the first 10 even numbers for now. The list comprehension we could use is [x*2 | x <- [1..10]] . x is drawn from [1..10] and for every element in [1..10] (which we have bound to x), we get that element, only doubled. Here's that comprehension in action.

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

```
listComprehension = [exp in x, y ... | domain of x, y, ..., predicate]
```

Exercise: Write a function that takes a string and removes everything except uppercase letters from it.

```
removeNonUppercase str = [ c | c <- str, c `elem` ['A'..'Z']]
```

Pattern matching on list

Here we match on the first argument to the infix (:) constructor, ignoring the rest of the list, and return that value:

```
Prelude> let myHead (x : _) = x
Prelude> :t myHead
myHead :: [t] -> t
Prelude> myHead [1, 2, 3]
1
```

We can do the opposite as well:

```
Prelude> let myTail (_ : xs) = xs
Prelude> :t myTail
myTail :: [t] -> [t]
Prelude> myTail [1, 2, 3]
[2,3]
```

We do need to be careful with functions like these. Neither myHead nor myTail has a case to handle an empty list — if we try to pass them an empty list as an argument, they can't pattern match:

```
Prelude> myHead []
*** Exception:
Non-exhaustive patterns
in function myHead
```

```
Prelude> myTail []
*** Exception:
Non-exhaustive patterns
in function myTail
```

Tuples

Tuples are used when you know exactly how many values you want to combine and its type depends on how many components it has and the types of the components. They are denoted with parentheses and their components are separated by commas. They don't have to be homogenous.Tuples can be used to represent a wide variety of data. For instance, if we wanted to represent someone's name and age in Haskell, we could use a triple: ("Christopher", "Walken", 55) .
Use tuples when you know in advance how many components some piece of data should have.Tuples are much more rigid because each different size of tuple is its own type, so you can't write a general function to append an element to a tuple.

Two useful functions that operate on pairs

fst

takes a pair and returns its first component.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

snd

takes a pair and returns its second component.

```
ghci> snd (8,11)
11
```

Note: these functions operate only on pairs. They won't work on triples, 4-tuples, 5-tuples, etc. We'll go over extracting data from tuples in different ways a bit later.

Type variable and Polymorphic functions

What do you think is the type of the head function? Because head takes a list of any type and returns the first element, so what could it be? Let's check!

```
ghci> :type head
head :: [a] -> a
```

What is this a? Is it a type? Remember that we previously stated that types are written in capital case, so it can't exactly be a type. Because it's not in capital case it's actually a **type variable**. That means that a can be of any type. This is much like generics in other languages, only in Haskell it's much more powerful because it allows us to easily write very

general functions if they don't use any specific behavior of the types in them. Functions that have type variables are called **polymorphic functions**. The type declaration of **head** states that it takes a list of any type and returns one element of that type.

Although type variables can have names longer than one character, we usually give them names of a, b, c, d …

Remember fst? It returns the first component of a pair. Let's examine its type.

```
ghci> :t fst
fst :: (a, b) -> a
```

We see that fst takes a tuple which contains two types and returns an element which is of the same type as the pair's first component. That's why we can use fst on a pair that contains any two types. Note that just because a and b are different type variables, they don't have to be different types. It just states that the first component's type and the return value's type are the same.

Custom types

Understanding the type system is a very important part of learning Haskell. A type is a kind of labels that every expression has. It tells us in which category of things that expression fits. The expression True is a boolean,“hello”is a string, etc.
So far, we've run into a lot of data types. Bool, Int, Char, etc. But how do we make our own?

Data declaration, Type constructors and Data constructors

We often want to create custom datatypes for structuring and describing the data we are processing. Doing so can help you analyze your problem by allowing you to focus first on how you model the domain before you begin thinking about how you write computations that solve your problem. It can also make your code easier to read and use because it lays the domain model out clearly.
In order to write your own types, though, you must under-stand the way datatypes are constructed in more detail than we’ve covered so far.

Activity:

1. In the REPL, run the commands
    a. :info Bool
    b. :info Maybe
    c. :info Either
2. What do you notice about the first line of the output?
    a. Run the command :type True
    b. Run the command :type Just
    c. Run the command :type Right
3. Can you write the general rule respected by this first line?

Lesson:

To create a new type in Haskell, we can use the so-called type declaration. The type declaration follow the following syntax:

```
data typeConstructor [typesArguments] = [ [dataConstructor1 [argumentsTypes1]] |  [dataConstructor2 [argumentsTypes2]] | ... | [dataConstructorn [argumentsTypesn]]]
```

We use:

- The keyword `data` to signal the compiler we are creating à new type. All what follows the keyword data is called data declaration or type declaration.
- **Type constructor**: This is the name of the new type we are creating. It alway begins with uppercase letters. It can have arguments written in lowercase alphabet characters. These arguments list are spaces separated.
- The equal (=) sign
- **Data constructors**: Can be present or not, and can have arguments or not. It alway begins with uppercase letters. Data constructor indicates how data of the type being defined is constructed. They are functions.

Example:
Data declaration with no argument
For Bool type the data declaration is:

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

- We see that the keyword data is used.
- The type constructor is Bool and it does not have any arguments
- We have two data constructors (True and False) with no arguments

Data declaration with one argument
For Maybe type the data declaration is:

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

- We see that the keyword data is used.
- The type constructor is Maybe and it takes one argument
- We have two data constructors (Nothing and Just). Nothing does not take any argument whereas Just takes one argument of type indicated by the type constructor argument.

Data declaration with two arguments
For Either type the data declaration is:

```
data Either a b = Left a | Right b
```

- We see that the keyword data is used.
- The type constructor is Either and it takes two arguments
- We have two data constructors (Left and Right). Left and Right take one argument of type indicated by the type constructor arguments.

Other Example:
We've been tasked with creating a data type that describes a person. The info that we want to store about that person is: full name, place of birth, year of birth, height, phone number. This is define as follow:

```
data Person = Person String String Int Float String deriving (Show)
let takam = Person "Takam Fridolin" "Yaounde" 1925 1.78 "691179855"
```

- This code declares a new type called `Person` .
- Person at the left of equal sign (=) is a type constructor with no argument.
- Person at the right of the equal sign (=) is a data constructor (à function) with five arguments of type String, String, Int, Float and String respectively.
- This type (Person) supports the typeclass Show. That means it implements the function `show :: a -> String`
- takam is data that has type Person.

Notes:
Types are static and resolved at compile time. Types are known before runtime, whether through explicit declaration or type inference, and that's what makes them static types. Information about types does not persist through to runtime. Data is what we're working with at runtime.
Here compile time is literally when your program is getting compiled by GHC or checked before execution in a REPL like GHCi. Runtime is the actual execution of your program. Types circumscribe values and in that way, they describe which values are flowing through what parts of your program.

type constructors ---- compile-time

---------- phase separation

data constructors ---- runtime

Now that we have a good understanding of the anatomy of datatypes, we want to start demonstrating why we call them "algebraic." We'll start by looking at something called arity. Arity refers to the number of arguments a function or constructor takes. A function that takes no arguments is called nullary, where nullary is a contraction of "null" and "-ary". Null means zero, the "-ary" suffix means "of or pertaining to". "-ary" is a common suffix used when talking about mathematical arity, such as with nullary, unary, binary, and the like.

Data constructors which take no arguments are also called nullary. Nullary data constructors, such as True and False, are constant values at the term level and, since they have no arguments, they can't construct or represent any data other than themselves. They are values which stand for themselves and act as a witness of the datatype they were declared in.
We've said that "A type can be thought of as an enumeration of constructors that have zero or more arguments."

Record syntax

What if we want to create a function to get separate info from a person? A function that gets some person's full name for instance.

```
getFullName :: Person -> String
getFullName (Person fullName placeOfBirth year heigh phone) = fullName
```

Here's how we could achieve the above functionality with record syntax.

```
data Person = Person {
     fullName :: String
   , placeOfBirth :: String
   , yearOfBirth :: Int
   , height :: Float
   , phoneNumber :: String
} deriving (Show)
```

So instead of just naming the field types one after another and separating them with spaces, we use curly brackets. First we write the name of the field, for instance, fullName and then we write a double colon : : and then we specify the type. The resulting data type is exactly the same. The main benefit of this is that it creates functions that lookup fields in the data type. By using record syntax to create this data type, Haskell automatically made these functions: *fullName, placeOfBirth, yearOfBirth, height, and phoneNumber.*

```
:t placeOfBirth
placeOfBirth :: Person -> String
```

There's another benefit to using record syntax. When we derive Show for the type, it displays it differently if we use record syntax to define and instantiate the type.

Type parameters

A data constructor can take some values parameters and then produce a new value. For instance, the Car constructor takes three values and produces a car value. In a similar manner, type constructors can take types as parameters to produce new types. This might sound a bit too meta at first, but it's not that complicated. To get a clear picture of what type parameters work like in action, let's take a look at how a type we've already met is implemented.

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

The a here is the type parameter. And because there's a type parameter involved, we call Maybe a type constructor. Depending on what we want this data type to hold when it's not Nothing, this type constructor can end up producing a type of Maybe Int, Maybe Car, Maybe String, etc. No value can have a type of just Maybe, because that's not a type per se, it's a type constructor. In order for this to be a real type that a value can be part of, it has to have all its type parameters filled up.
So if we pass Char as the type parameter to Maybe, we get a type of Maybe Char. The value Just 'a' has a type of Maybe Char, for example.

You might not know it, but we used a type that has a type parameter before we used Maybe. That type is the list type. Although there's some syntactic sugar in play, the list type takes a

parameter to produce a concrete type. Values can have an [Int] type, a [Char] type, a [[String]] type, but you can't have a value that just has a type of [].
Let's play around with the Maybe type.

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

Type parameters are useful because we can make different types with them depending on what kind of types we want contained in our data type. When we do :t Just "Haha", the type inference engine figures it out to be of the type Maybe [Char], because if the a in the Just a is a string, then the a in Maybe a must also be a string.

Notice that the type of Nothing is Maybe a. Its type is polymorphic. If some function requires a Maybe Int as a parameter, we can give it a Nothing, because a Nothing doesn't contain a value anyway and so it doesn't matter. The Maybe a type can act like a Maybe Int if it has to, just like 5 can act like an Int or a Double. Similarly, the type of the empty list is [a]. An empty list can act like a list of anything (type).
Using type parameters is very beneficial, but only when using them makes sense. Usually we use them when our data type would work regardless of the type of the value it then holds inside it, like with our Maybe a type. If our type acts as some kind of box, it's good to use them. We could change our Car data type from this:

```
data Car = Car {
   company :: String
   , model :: String
   , year :: Int
} deriving (Show)
```

To this:

```
data Car a b c = Car {
   company :: a
   , model :: b
   , year :: c
} deriving (Show)
```

But would we really benefit? The answer is: probably no, because we'd just end up defining functions that only work on the Car String String Int type. For instance, given our first definition of Car, we could make a function that displays the car's properties in a nice little text.

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " "
++ m ++ " was made in " ++ show y
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

A cute little function! The type declaration is cute and it works nicely. Now what if Car was Car a b c?

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " "
++ m ++ " was made in " ++ show y
```

We'd have to force this function to take a Car type of (Show a) => Car String String a. You can see that the type signature is more complicated and the only benefit we'd actually get would be that we can use any type that's an instance of the Show typeclass as the type for c.

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

In real life though, we'd end up using Car String String Int most of the time and so it would seem that parameterizing the Car type isn't really worth it. We usually use type parameters when the type that's contained inside the data type's various value constructors isn't really that important for the type to work. A list of stuff is a list of stuff and it doesn't matter what the type of that stuff is, it can still work. If we want to sum a list of numbers, we can specify later in the summing function that we specifically want a list of numbers. Same goes for Maybe. Maybe represents an option of either having nothing or having one of something. It doesn't matter what the type of that something is.

Another example of a parameterized type is Map k v from Data.Map. The k is the type of the keys in a map and the v is the type of the values. This is a good example of where type parameters are very useful. Having maps parameterized enables us to have mappings from any type to any other type, as long as the type of the key is part of the Ord typeclass. If we were defining a mapping type, we could add a typeclass constraint in the data declaration:

```
data (Ord k) => Map k v = ...
```

However, it's a very strong convention in Haskell to never add typeclass constraints in data declarations. Why? Well, because we don't benefit a lot, but we end up writing more class constraints, even when we don't need them. If we put or don't put the Ord k constraint in the data declaration for Map k v, we're going to have to put the constraint into functions that assume the keys in a map can be ordered. But if we don't put the constraint in the data declaration, we don't have to put (Ord k) => in the type declarations of functions that don't care whether the keys can be ordered or not. An example of such a function is toList, that

just takes a mapping and converts it to an associative list. Its type signature is toList :: Map k a -> [(k, a)]. If Map k v had a type constraint
in its data declaration, the type for toList would have to be toList :: (Ord k) => Map k a -> [(k, a)], even though the function doesn't do any comparison of keys by order. So don't put type constraints into data declarations even if it seems to make sense, because you'll have to put them into the function type declarations either way.

Let's implement a 3D vector type and add some operations for it. We'll be using a parameterized type because even though it will usually contain numeric types, it will still support several of them.

```
data Vector a = Vector a a a deriving (Show)
vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)
vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)
scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

vplus is for adding two vectors together. Two vectors are added just by adding their corresponding components. scalarMult is for the scalar product of two vectors and vectMult is for multiplying a vector with a scalar. These functions can operate on types of Vector Int, Vector Integer, Vector Float, whatever, as long as the a from Vector a is from the Num typeclass. Also, if you examine the type declaration for these functions, you'll see that they can operate only on vectors of the same type and the numbers involved must also be of the type that is contained in the vectors. Notice that we didn't put a Num class constraint in the data declaration, because we'd have to repeat it in the functions anyway. Once again, it's very important to distinguish between the type constructor and the value constructor. When declaring a data type, the part before the = is the type constructor and the constructors after it (possibly separated by |'s) are value constructors or data constructor. Giving a function a type of `Vector t t t -> Vector t t t -> t` would be wrong, because we have to put types in type declaration and the vector type constructor takes only one parameter, whereas the value constructor takes three. Let's play around with our vectors.

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

Algebraic data types (Maybe, Either, etc)

A type can be thought of as an enumeration of constructors that have zero or more arguments. Haskell offers sum types, product types, product types with record syntax, type aliases (for example, String is a type alias for [Char]), and a special datatype called a newtype that provides for a different set of options and constraints from either type synonyms or data declarations.

Type synonyme

Previously, we mentioned that when writing types, the [Char] and String types are equivalent and interchangeable. That's implemented with type synonyms. Type synonyms don't really do anything per se, they're just about giving some types different names so that they make more sense to someone reading our code and documentation. Here's how the standard library defines String as a synonym for [Char].

```
type String = [Char]
```

We've introduced the **type** keyword. The keyword might be misleading to some, because we're not actually making anything new (we did that with the data keyword), but we're just making a synonym for an already existing type. If we make a function that converts a string to uppercase and call it toUpperString or something, we can give it a type declaration of

```
toUpperString : [Char] -> [Char]
or toUpperString : : String -> String
```

Both of these are essentially the same, only the latter is nicer to read.

Type wrapper (newtype syntax)

The newtype keyword in Haskell is made exactly when we want to just take one type and wrap it in something to present it as another type.
A newtype declaration creates a new type in much the same way as data. The syntax and usage of newtypes is virtually identical to that of data declarations - in fact, you can replace the newtype keyword with data and it'll still compile, indeed there's even a good chance your program will still work. The converse is not true, however - data can only be replaced with newtype if the type has exactly one constructor with exactly one field inside it.

Example:

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

Recursive data structure

As we've seen, a constructor in an algebraic data type can have several (or none at all) fields and each field must be of some concrete type. With that in mind, we can make types whose constructors have fields that are of the same type. Using that, we can create recursive data types, where one value of some type contains values of that type, which in turn contain more values of the same type and so on.

List definition with cons (:)

Cons operator

Activity

1. In the Haskell REPL, run the command `:type (:)`
2. Run the command `(:) 2 [7, 4]` or `2 : [7, 4]`
3. What does the (:) do?
4. Run the command `2 : (7 : (4 : []))`
5. Run the command `:info []` and how do you interpret the result

Think about this list: [5] . That's just syntactic sugar for 5 : [ ] . On the left side of the : , there's a value and on the right side, there's a list. And in this case, it's an empty list. Now how about the list [4,5]? Well, that desugars to 4:(5:[]). Looking at the first :, we see that it also has an element on its left side and a list (5:[]) on its right side. Same goes for a list like 3:(4:(5:6:[])), which could be written either like that or like 3:4:5:6:[] (because : is right-associative) or [3,4,5,6].

Definition of list

We could say that a list can be an empty list or it can be an element joined together with a : with another list (that can be either the empty list or not).
Let's use algebraic data types to implement our own list then!

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

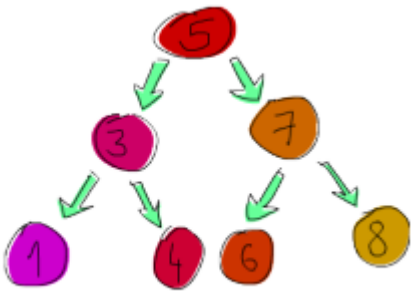
A list is either an empty list or a combination of a head with some value and a list.
You might also be confused about the Cons constructor here. cons is another word for : .
You see, in lists, : is actually a constructor that takes a value and another list and returns a list. We can already use our new list type! In other words, it has two fields. One field is of the type of a and the other is of the type [a].

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

We called our Cons constructor in an infix manner so you can see how it's just like : . Empty is like [] and 4 `Cons` (5 `Cons` Empty) is like 4:(5:[]).

binary tree

Graphical representation



Description

If you're not familiar with binary search trees from languages like C, here's what they are: an element points to two elements, one on its left and one on its right. The element to the left is smaller, the element to the right is bigger. Each of those elements can also point to two elements (or one, or none). In effect, each element has up to two sub-trees. And a cool thing about binary search trees is that we know that all the elements at the left subtree of, say, 5 are going to be smaller than 5. Elements in its right subtree are going to be bigger. So if we need to find if 8 is in our tree, we'd start at 5 and then because 8 is greater than 5, we'd go right. We're now at 7 and because 8 is greater than 7, we go right again. And we've found our element in three hops! Now if this were a normal list (or a tree, but really unbalanced), it would take us seven hops instead of three to see if 8 is in there. Sets and maps from Data.Set and Data.Map are implemented using trees, only instead of normal binary search trees, they use balanced binary search trees, which are always balanced. But right now, we'll just be implementing normal binary search trees.

Definition of binary tree

Here's what we're going to say: a tree is either an empty tree or it's an element that contains some value and two trees. Sounds like a perfect fit for an algebraic data type!

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Activity

1. Let's make a function that takes a tree and an element and inserts the element in the tree.
2. Write a function that checks if a given element is in the tree.

Solution

1- We do this by comparing the value we want to insert to the root node and then if it's smaller, we go left, if it's larger, we go right. We do the same for every subsequent node until we reach an empty tree. Once we've reached an empty tree, we just insert a node with that value instead of the empty tree.
As in Haskell, we can't really modify our tree, so we have to make a new sub-tree each time we decide to go left or right and in the end the insertion function returns a completely new

tree, because Haskell respect the immutability of data. Hence, the type for our insertion function is going to be something like a -> Tree a - > Tree a. It takes an element and a tree and returns a new tree that has that element inside. This might seem like it's inefficient but laziness takes care of that problem.
So, here is the function.

```
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = Node x EmptyTree EmptyTree
treeInsert x (Node a left right)
   | x == a = Node x left right
   | x < a = Node a (treeInsert x left) right
   | x > a = Node a left (treeInsert x right)
```

In the insertion function, we first have the edge condition as a pattern. If we've reached an empty subtree, that means we're where we want and instead of the empty tree, we create a node with our element. If we're not inserting into an empty tree, then we have to check some things. First off, if the element we're inserting is equal to the root element, just return a tree that's the same. If it's smaller, return a tree that has the same root value, the same right subtree but instead of its left subtree, put a tree that has our value inserted into it. Same (but the other way around) goes if our value is bigger than the root element.

2- To check that an element is in à tree, we first define the edge condition. If we're looking for an element in an empty tree, then it's certainly not there. Anyway, if we're looking for an element in a non-empty tree, then we check some things. If the element in the root node is what we're looking for, great! If it's not, what then? Well, we can take advantage of knowing that all the left elements are smaller than the root node. So if the element we're looking for is smaller than the root node, check to see if it's in the left subtree. If it's bigger, check to see if it's in the right subtree.

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a = treeElem x left
    | x > a = treeElem x right
```

Kind

We saw that a type is defined with a type constructor. The type constructor can take parameters. When a type constructor takes a parameter, it works as a function that takes as input a type and returns another type. For example Maybe. We've seen that type constructors can be partially applied (Either String is a type that takes one type and produces a concrete type, like Either String Int), just like functions can.
Kinds are the types of type constructors. Kind signatures work like type signatures, using the same :: and -> syntax, but there are only a few kinds and you'll most often see *.
To query the kind of a type constructor we use the command `:kind` or `:k`

Let's examine the kind of a type by using the :k command in GHCI.

```
ghci> :k Int
Int :: *
```

This means that Int is à concret type.

```
ghci> :k Maybe
Maybe :: * -> *
```

This means that Maybe takes one concrete type (like Int) and then returns a concrete type like Maybe Int. And that's what this kind tells us. Just like Int -> Int means that a function takes an Int and returns an Int, * -> * means that the type constructor takes one concrete type and returns a concrete type. Let's apply the type parameter to Maybe and see what the kind of that type is.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Other Example

```
data Silly a b c d = MkSilly a b c d deriving Show

ghci> :kind Silly
Silly :: * -> * -> * -> * -> *

ghci> :kind Silly Int
Silly Int :: * -> * -> * -> *

ghci> :kind Silly Int String
Silly Int String :: * -> * -> *

ghci> :kind Silly Int String Bool
Silly Int String Bool :: * -> *

ghci> :kind Silly Int String Bool String
Silly Int String Bool String :: *
```

```
ghci> :kind (,,,)
(,,,) :: * -> * -> * -> * -> *

ghci> :kind (Int, String, Bool, String)
(Int, String, Bool, String) :: *
```