Learn You a Haskell for Great Good!

Miran Lipovača

2

Contents

Chapter 1

Introduction

1.1 About this tutorial

Welcome to **Learn You a Haskell for Great Good**! If you're reading this, chances are you want to learn Haskell. Well, you've come to the right place, but let's talk about this tutorial a bit first.

I decided to write this because I wanted to solidify my own knowledge of Haskell and because I thought I could help people new to Haskell learn it from my perspective. There are quite a few tutorials on Haskell floating around on the internet. When I was starting out in Haskell, I didn't learn from just one resource. The way I learned it was by reading several different tutorials and articles because each explained something in a different way than the other did. By going through several resources, I was able put together the pieces and it all just came falling into place. So this is an attempt at adding another useful resource for learning Haskell so you have a bigger chance of finding one you like.

This tutorial is aimed at people who have experience in imperative programming languages (C, C++, Java, Python ...) but haven't programmed in a functional language before (Haskell, ML, OCaml ...). Although I bet that even if you don't have any significant programming experience, a smart chap like you will be able to follow along and learn Haskell.

The channel #haskell on the freenode network is a great place to ask questions if you're feeling stuck. People there are extremely nice, patient and understanding to newbies.

I failed to learn Haskell approximately 2 times before finally grasping it because it all just seemed too weird to me and I didn't get it. But then once it just "clicked" and after getting over that initial hurdle, it was pretty much smooth sailing. I guess what I'm trying to say is: Haskell is great and if you're interested in programming you should really learn it even if it seems weird at first. Learning Haskell is much like learning to program for the first time — it's fun! It forces you to think differently, which brings us to the next section ...

1.2   So what's Haskell?

Haskell is a **purely functional programming language**. In imperative languages you get things done by giving the computer a sequence of tasks and then it executes them. While executing them, it can change state. For instance, you set variable `a` to 5 and then do some stuff and then set it to something else. You have control flow structures for doing some action several times. In purely functional programming you don't tell the computer what to do as such but rather you tell it what stuff is. The factorial of a number is the product of all the numbers from 1 to that number, the sum of a list of numbers is the first number plus the sum of all the other numbers, and so on. You express that in the form of functions. You also can't set a variable to something and then set it to something else later. If you say that `a` is 5, you can't say it's something else later because you just said it was 5. What are you, some kind of liar? So in purely functional languages, a function has no side-effects. The only thing a function can do is calculate something and return it as a result. At first, this seems kind of limiting but it actually has some very nice consequences: if a function is called twice with the same parameters, it's guaranteed to return the same result. That's called referential transparency and not only does it allow the compiler to reason about the program's behavior, but it also allows you to easily deduce (and even prove) that a function is correct and then build more complex functions by gluing simple functions together.

Haskell is **lazy**. That means that unless specifically told otherwise, Haskell won't execute functions and calculate things until it's really forced to show you a result. That goes well with referential transparency and it allows you to think of programs as a series of **transformations on data**. It also allows cool things such as infinite data structures. Say you have an immutable list of numbers `xs = [1,2,3,4,5,6,7,8]` and a function `doubleMe` which multiplies every element by 2 and then returns a new list. If we wanted to multiply our list by 8 in an imperative language and did `doubleMe(doubleMe(doubleMe(xs)))`, it would probably pass through the list once and make a copy and then return it. Then it would pass through the list another two times and return the result. In a lazy language, calling `doubleMe` on a list without forcing it to show you the result ends up in the program sort of telling you "Yeah yeah, I'll do it later!". But once you want to see the result, the first `doubleMe` tells the second one it wants the result, now! The second one says that to the third one and the third one reluctantly gives back a doubled 1, which is a 2. The second one receives that and gives back 4 to the first one. The first one sees that and tells you the first element is 8. So it only does one pass through the list and only when you really need it. That way when you want something from a lazy language you can just take some initial data and efficiently transform and mend it so it resembles what you want at the end.

Haskell is **statically typed**. When you compile your program, the compiler knows which piece of code is a number, which is a string and so on. That means that a lot of possible errors are caught at compile time. If you try to add together a number and a string, the compiler will whine at you. Haskell uses a very good type system that has **type inference**. That means that you don't have to explicitly label every piece of code with a type because the type

system can intelligently figure out a lot about it. If you say `a = 5 + 4`, you don't have to tell Haskell that `a` is a number, it can figure that out by itself. Type inference also allows your code to be more general. If a function you make takes two parameters and adds them together and you don't explicitly state their type, the function will work on any two parameters that act like numbers.

Haskell is **elegant and concise**. Because it uses a lot of high level concepts, Haskell programs are usually shorter than their imperative equivalents. And shorter programs are easier to maintain than longer ones and have less bugs.

Haskell was made by some **really smart guys** (with PhDs). Work on Haskell began in 1987 when a committee of researchers got together to design a kick-ass language. In 2003 the Haskell Report was published, which defines a stable version of the language.

1.3   What you need to dive in

A text editor and a Haskell compiler. You probably already have your favorite text editor installed so we won't waste time on that. The two main Haskell compilers as of right now are GHC (Glasgow Haskell Compiler) and Hugs. For the purposes of this tutorial we'll be using GHC. I won't cover much installation details. On Windows it's just a matter of downloading the installer and clicking "Next" a couple of times and then rebooting your computer. On Debian based Linux distributions you can just do `apt-get install ghc6 libghc6-mtl-dev` and you're laughing. I don't have a Mac but I've heard that if you have MacPorts, you can get GHC by doing `sudo port install ghc`. Also, I think you can do Haskell development with that wacky mouse with one button, although I'm not sure.

GHC can take a Haskell script (they usually have a .hs extension) and compile it but it also has an interactive mode which allows you to interactively interact with scripts. Interactively. You can call functions from scripts that you load and the results are displayed immediately. For learning it's a lot easier and faster than compiling every time you make a change and then running the program from the prompt. The interactive mode is invoked by typing in `ghci` at your prompt. If you have defined some functions in a file called, say, `myfunctions.hs`, you load up those functions by typing in `:l myfunctions` and then you can play with them, provided `myfunctions.hs` is in the same folder from which `ghci` was invoked. If you change the .hs script, just run `:l myfunctions` again or do `:r`, which is equivalent because it reloads the current script. The usual workflow for me when playing around in stuff is defining some functions in a .hs file, loading it up and messing around with them and then changing the .hs file, loading it up again and so on. This is also what we'll be doing here.

Chapter 2

Starting Out

2.1 Ready, set, go!

Alright, let's get started! If you're the sort of horrible person who doesn't read introductions to things and you skipped it, you might want to read the last section in the introduction anyway because it explains what you need to follow this tutorial and how we're going to load functions. The first thing we're going to do is run ghc's interactive mode and call some function to get a very basic feel for haskell. Open your terminal and type in `ghci`. You will be greeted with something like this.

```
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Congratulations, you're in GHCI! The prompt here is `Prelude>` but because it can get longer when you load stuff into the session, we're going to use `ghci>`. If you want to have the same prompt, just type in `:set prompt "ghci> "`.

Here's some simple arithmetic.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

This is pretty self-explanatory. We can also use several operators on one line and all the usual precedence rules are obeyed. We can use parentheses to make the precedence explicit or to change it.

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

9

Pretty cool, huh? Yeah, I know it's not but bear with me. A little pitfall to watch out for here is negating numbers. If we want to have a negative number, it's always best to surround it with parentheses. Doing `5 * -3` will make GHCI yell at you but doing `5 * (-3)` will work just fine.

Boolean algebra is also pretty straightforward. As you probably know, `&&` means a boolean and, `||` means a boolean or. `not` negates a `True` or a `False`.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Testing for equality is done like so.

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

What about doing `5 + "llama"` or `5 == True`? Well, if we try the first snippet, we get a big scary error message!

```
No instance for (Num [Char])
arising from a use of '+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of 'it': it = 5 + "llama"
```

Yikes! What GHCI is telling us here is that `"llama"` is not a number and so it doesn't know how to add it to 5. Even if it wasn't `"llama"` but `"four"` or `"4"`, Haskell still wouldn't consider it to be a number. `+` expects its left and right side to be numbers. If we tried to do `True == 5`, GHCI would tell us that the types don't match. Whereas `+` works only on things that are considered numbers, `==` works on any two things that can be compared. But the catch is that they both have to be the same type of thing. You can't compare apples and oranges. We'll take a closer look at types a bit later. Note: you can do `5 + 4.0` because `5` is sneaky and can act like an integer or a floating-point number. `4.0` can't act like an integer, so `5` is the one that has to adapt.

You may not have known it but we've been using functions now all along. For instance, `*` is a function that takes two numbers and multiplies them. As you've seen, we call it by sandwiching it between them. This is what we call an *infix* function. Most functions that aren't used with numbers are *prefix* functions. Let's take a look at them.

Functions are usually prefix so from now on we won't explicitly state that a function is of the prefix form, we'll just assume it. In most imperative languages functions are called by writing the function name and then writing its parameters in parentheses, usually separated by commas. In Haskell, functions are called by writing the function name, a space and then the parameters, separated by spaces. For a start, we'll try calling one of the most boring functions in Haskell.

```
ghci> succ 8
9
```

The `succ` function takes anything that has a defined successor and returns that successor. As you can see, we just separate the function name from the parameter with a space. Calling a function with several parameters is also simple. The functions `min` and `max` take two things that can be put in an order (like numbers!) and return the one that's greater.

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

Function application (calling a function by putting a space after it and then typing out the parameters) has the highest precedence of them all. What that means for us is that these two statements are equivalent.

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

However, if we wanted to get the successor of the product of numbers 9 and 10, we couldn't write `succ 9 * 10` because that would get the successor of 9, which would then be multiplied by 10. So 100. We'd have to write `succ (9 * 10)` to get 91.

If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks. For instance, the `div` function takes two integers and does integral division between them. Doing `div 92 10` results in a 9. But when we call it like that, there may be some confusion as to which number is doing the division and which one is being divided. So we can call it as an infix function by doing `92 `div` 10` and suddenly it's much clearer.

Lots of people who come from imperative languages tend to stick to the notion that parentheses should denote function application. For example, in C, you use parentheses to call functions like `foo()`, `bar(1)` or `baz(3, "haha")`. Like we said, spaces are used for function application in Haskell. So those functions in Haskell would be `foo`, `bar 1` and `baz 3 "haha"`. So if you see something like `bar (bar 3)`, it doesn't mean that `bar` is called with `bar` and `3` as parameters. It means that we first call the function `bar` with `3` as the parameter to get some number and then we call `bar` again with that number. In C, that would be something like `bar(bar(3))`.

2.2   Baby's first functions

In the previous section we got a basic feel for calling functions. Now let's try making our own! Open up your favorite text editor and punch in this function that takes a number and multiplies it by two.

```
doubleMe x = x + x
```

Functions are defined in a similar way that they are called. The function name is followed by parameters seperated by spaces. But when defining functions, there's a = and after that we define what the function does. Save this as `baby.hs` or something. Now navigate to where it's save and run `ghci` from there. Once inside GHCI, do `:l baby`. Now that our script is loaded, we can play with the function that we defined.

```
ghci> :l baby
[1 of 1] Compiling Main              ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Because + works on integers as well as on floating-point numbers (anything that can be considered a number, really), our function also works on any number. Let's make a function that takes two numbers and multiplies each by two and then adds them together.

```
doubleUs x y = x*2 + y*2
```

Simple. We could have also defined it as `doubleUs x y = x + x + y + y`. Testing it out produces pretty predictable results (remember to append this function to the `baby.hs` file, save it and then do `:l baby` inside GHCI).

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

As expected, you can call your own functions from other functions that you made. With that in mind, we could redefine `doubleUs` like this:

```
doubleUs x y = doubleMe x + doubleMe y
```

This is a very simple example of a common pattern you will see throughout Haskell. Making basic functions that are obviously correct and then combining them into more complex functions. This way you also avoid repetition. What if some mathematicians figured out that 2 is actually 3 and you had to change your program? You could just redefine `doubleMe` to be `x + x + x` and since `doubleUs` calls `doubleMe`, it would automatically work in this strange new world where 2 is 3.

Functions in Haskell don't have to be in any particular order, so it doesn't matter if you define `doubleMe` first and then `doubleUs` or if you do it the other way around.

Now we're going to make a function that multiplies a number by 2 but only if that number is smaller than or equal to 100 because numbers bigger than 100 are big enough as it is!

```
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
```

Right here we introduced Haskell's if statement. You're probably familiar with if statements from other languages. The difference between Haskell's if statement and if statements in imperative languages is that the else part is mandatory in Haskell. In imperative languages you can just skip a couple of steps if the condition isn't satisfied but in Haskell every expression and function must return something. We could have also written that if statement in one line but I find this way more readable. Another thing about the if statement in Haskell is that it is an *expression*. An expression is basically a piece of code that returns a value. `5` is an expression because it returns 5, `4 + 8` is an expression, `x + y` is an expression because it returns the sum of `x` and `y`. Because the else is mandatory, an if statement will always return something and that's why it's an expression. If we wanted to add one to every number that's produced in our previous function, we could have written its body like this.

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Had we ommitted the parentheses, it would have added one only if `x` wasn't greater than 100. Note the ' at the end of the function name. That apostrophe doesn't have any special meaning in Haskell's syntax. It's a valid character to use in a function name. We usually use ' to either denote a strict version of a function (one that isn't lazy) or a slightly modified version of a function or a variable. Because ' is a valid character in functions, we can make a function like this.

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

There are two noteworthy things here. The first is that in the function name we didn't capitalize Conan's name. That's because functions can't begin with uppercase letters. We'll see why a bit later. The second thing is that this function doesn't take any parameters. When a function doesn't take any parameters, we usually say it's a *definition* (or a *name*). Because we can't change what names (and functions) mean once we've defined them, `conanO'Brien` and the string `"It's a-me, Conan O'Brien!"` can be used interchangeably.

2.3 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. And now, a list!

> **Note**: We can use the `let` keyword to define a name right in GHCI.
> Doing `let a = 1` inside GHCI is the equivalent of writing `a = 1` in a
> script and then loading it.

```
ghci> let lostNumbers = [4,8,15,16,23,48]
ghci> lostNumbers
[4,8,15,16,23,48]
```

As you can see, lists are denoted by square brackets and the values in the lists are separated by commas. 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. Speaking of characters, strings are just lists of characters. `"hello"` is just syntactic sugar for `['h','e','l','l','o']`. Because strings are lists, we can use list functions on them, which is really handy.

A common task is putting two lists together. This is done by using the `++` operator.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Watch out when repeatedly using the `++` operator on long strings. When you put together two lists (even if you append a singleton list to a list, for instance: `[1,2,3] ++ [4]`), internally, Haskell has to walk through the whole list on the left side of `++`. That's not a problem when dealing with lists that aren't too big. But putting something at the end of a list that's fifty million entries long is going to take a while. However, putting something at the beginning of a list using the `:` operator (also called the cons operator) is instantaneous.

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Notice how `:` takes a number and a list of numbers or a character and a list of characters, whereas `++` takes two lists. Even if you're adding an element to the end of a list with `++`, you have to surround it with square brackets so it becomes a list.

`[1,2,3]` is actually just syntactic sugar for `1:2:3:[]`. `[]` is an empty list. If we prepend `3` to it, it becomes `[3]`. If we prepend `2` to that, it becomes `[2,3]`, and so on.

> **Note**: `[]`, `[[]]` and `[[],[],[]]` are all different things. The first one
> is an empty list, the seconds one is a list that contains one empty
> list, the third one is a list that contains three empty lists.

If you want to get an element out of a list by index, use `!!`. The indices start at 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

But if you try to get the sixth element from a list that only has four elements, you'll get an error so be careful!

Lists can also contain lists. They can also contain lists that contain lists that contain lists . . .

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

The lists within a list can be of different lengths but they can't be of different types. Just like you can't have a list that has some characters and some numbers, you can't have a list that has some lists of characters and some lists of numbers.

Lists can be compared if the stuff they contain can be compared. When using <, <=, > and >= to compare lists, they are compared in lexicographical order. First the heads are compared. If they are equal then the second elements are compared, etc.

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

What else can you do with lists? Here are some basic functions that operate on lists.

`hail` 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` returns the smallest.

```
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
```

Those were a few basic functions that operate on lists. We'll take a look at more list functions in section #TODO: include section name and link here.

2.4 Texas ranges

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 1 and 20? Or every third number between 1 and 20?

```
ghci> [1..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]
```

My advice is not 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:

`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]`.

2.5   I'm a 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 $S = \{2 \cdot x | x \in \mathbb{N}, x \leq 10\}$. The part before the pipe is called the output function, `x` is the variable, `N` is the input set and `x <= 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]
```

As you can see, we get the desired results. Now let's add a condition (or a predicate) to that comprehension. Predicates go after the binding parts and are separated from them by a comma. Let's say we want only the elements which, doubled, are greater than or equal to 12.

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

Cool, it works. How about if we wanted all numbers from 50 to 100 whose remainder when divided with the number 7 is 3? Easy.

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

Success! Note that weeding out lists by predicates is also called **filtering**. We took a list of numbers and we filtered them by the predicate. Now for another example. Let's say we want a comprehension that replaces each odd number greater than 10 with `"BANG!"` and each odd number that's less than 10 with `"BOOM!"`. If a number isn't odd, we throw it out of our list. For convenience, we'll put that comprehension inside a function so we can easily reuse it.

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

The last part of the comprehension is the predicate. The function `odd` returns `True` on an odd number and `False` on an even one. The element is included in the list only if all the predicates evaluate to `True`.

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

We can include several predicates. If we wanted all numbers from 10 to 20 that are not 13, 15 or 19, we'd do:

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Not only can we have multiple predicates in list comprehensions (an element must satisfy all the predicates to be included in the resulting list), we can also draw from several lists. When drawing from several lists, comprehensions produce all combinations of the given lists and then join them by the output

function we supply. A list produced by a comprehension that draws from two
lists of length 4 will have a length of 16, provided we don't filter them. If we
have two lists, `[2,5,10]` and `[8,10,11]` and we want to get the products of all
the possible combinations between numbers in those lists, here's what we'd
do.

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

As expected, the length of the new list is 9. What if we wanted all possible
products that are more than 50?

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

How about a list comprehension that combines a list of adjectives and a
list of nouns . . . for epic hilarity.

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

I know! Let's write our own version of `length`! We'll call it `length'`.

```
length' xs = sum [1 | _ <- xs]
```

`_` means that we don't care what we'll draw from the list anyway so instead
of writing a variable name that we'll never use, we just write `_`. This function
replaces every element of a list with `1` and then sums that up. This means
that the resulting sum will be the length of our list.

Just a friendly reminder: because strings are lists, we can use list compre-
hensions to process and produce strings. Here's a function that takes a string
and removes everything except uppercase letters from it.

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

Testing it out:

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

The predicate here does all the work. It says that the character will be
included in the new list only if it's an element of the list `['A'..'Z']`. Nested
list comprehensions are also possible if you're operating on lists that contain
lists. A list contains several lists of numbers. Let's remove all odd numbers
without flattening the list.

```
ghci> let xxs =
[[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

You can write list comprehensions across several lines. So if you're not
in GHCI, it's better to split longer list comprehensions across multiple lines,
especially if they're nested.

2.6 Tuples

In some ways, tuples are like lists — they are a way to store several values into
a single value. However, there are a few fundamental differences. A list of
numbers is a list of numbers. That's its type and it doesn't matter if it has only
one number in it or an infinite amount of numbers. Tuples, however, 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.

Another key difference is that they don't have to be homogenous. Unlike a
list, a tuple can contain a combination of several types.

Think about how we'd represent a two-dimensional vector in Haskell. One
way would be to use a list. That would kind of work. So what if we wanted
to put a couple of vectors in a list to represent points of a shape on a two-
dimensional plane? We could do something like `[[1,2],[8,11],[4,5]]`. The prob-
lem with that method is that we could also do stuff like `[[1,2],[8,11,5],[4,5]]`,
which Haskell has no problem with since it's still a list of lists with numbers
but it kind of doesn't make sense. But a tuple of size two (also called a pair) is
its own type, which means that a list can't have a couple of pairs in it and then
a triple (a tuple of size three), so let's use that instead. Instead of surrounding
the vectors with square brackets, we use parentheses: `[(1,2),(8,11),(4,5)]`.
What if we tried to make a shape like `[(1,2),(8,11,5),(4,5)]`? Well, we'd get
this error:

```
Couldn't match expected type '(t, t1)'
against inferred type '(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of 'it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

It's telling us that we tried to use a pair and a triple in the same list, which
is not supposed to happen. You also couldn't make a list like `[(1,2),("One",2)]`
because the first element of the list is a pair of numbers and the second element
is a pair consisting of a string and a number. Tuples can also be used to repre-
sent 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)`.
As seen in this example, tuples can also contain lists.

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 — you'd have to write a function for appending to a pair,
one function for appending to a triple, one function for appending to a 4-tuple,
etc.

While there are singleton lists, there's no such thing as a singleton tuple.
It doesn't really make much sense when you think about it. A singleton tu-
ple would just be the value it contains and as such would have no benefit to us.

Like lists, tuples can be compared with each other if their components can be compared. Only you can't compare two tuples of different sizes, whereas you can compare two lists of different sizes. 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. Surprise!

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

> **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.

A cool function that produces a list of pairs: `zip`. It takes two lists and then zips them together into one list by joining the matching elements into pairs. It's a really simple function but it has loads of uses. It's especially useful for when you want to combine two lists in a way or traverse two lists simultaneously. Here's a demonstration.

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

It pairs up the elements and produces a new list. The first element goes with the first, the second with the second, etc. Notice that because pairs can have different types in them, `zip` can take two lists that contain different types and zip them up. What happens if the lengths of the lists don't match?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

The longer list simply gets cut off to match the length of the shorter one. Because Haskell is lazy, we can zip finite lists with infinite lists:

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Here's a problem that combines tuples and list comprehensions: which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24? First, let's try generating all triangles with sides equal to or smaller than 10:

```
ghci> let triangles = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
```

We're just drawing from three lists and our output function is combining them into a triple. If you evaluate that by typing out `triangles` in GHCI, you'll get a list of all possible triangles with sides under or equal to 10. Next, we'll

add a condition that they all have to be right triangles. We'll also modify
this function by taking into consideration that side b isn't larger than the
hypothenuse and that side a isn't larger than side b.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <-
[1..b], a^2 + b^2 == c^2]
```

We're almost done. Now, we just modify the function by saying that we
want the ones where the perimeter is 24.

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <-
[1..b], a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

And there's our answer! This is a common pattern in functional program-
ming. You take a starting set of solutions and then you apply transformations
to those solutions and filter them until you get the right ones.

Chapter 3

Types and Typeclasses

3.1 Believe the type

Previously we mentioned that Haskell has a static type system. The type of every expression is known at compile time, which leads to safer code. If you write a program where you try to divide a boolean type with some number, it won't even compile. That's good because it's better to catch such errors at compile time instead of having your program crash. Everything in Haskell has a type, so the compiler can reason quite a lot about your program before compiling it.

Unlike Java or Pascal, Haskell has type inference. If we write a number, we don't have to tell Haskell it's a number. It can *infer* that on its own, so we don't have to explicitly write out the types of our functions and expressions to get things done. We covered some of the basics of Haskell with only a very superficial glance at types. However, understanding the type system is a very important part of learning Haskell.

A type is a kind of label 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.

Now we'll use GHCI to examine the types of some expressions. We'll do that by using the `:t` command which, followed by any valid expression, tells us its type. Let's give it a whirl.

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Here we see that doing `:t` on an expression prints out the expression followed by `::` and its type. `::` is read as "has type of". Explicit types are

always denoted with the first letter in capital case. `'a'`, as it would seem, has a type of `Char`. It's not hard to conclude that it stands for *character*. `True` is of a `Bool` type. That makes sense. But what's this? Examining the type of `"HELLO!"` yields a `[Char]`. The square brackets denote a list. So we read that as it being *a list of characters*. Unlike lists, each tuple length has its own type. So the expression of `(True, 'a')` has a type of `(Bool, Char)`, whereas an expression such as `('a','b','c')` would have the type of `(Char, Char, Char)`. `4 == 5` will always return `False`, so its type is `Bool`.

Functions also have types. When writing our own functions, we can choose to give them an explicit type declaration. This is generally considered to be good practice except when writing very short functions. From here on, we'll give all the functions that we make type declarations. Remember the list comprehension we made previously that filters a string so that only caps remain? Here's how it looks like with a type declaration.

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` has a type of `[Char] -> [Char]`, meaning that it maps from a string to a string. That's because it takes one string as a parameter and returns another as a result. The `[Char]` type is synonymous with `String` so it's clearer if we write `removeNonUppercase :: String -> String`. We didn't have to give this function a type declaration because the compiler can infer by itself that it's a function from a string to a string but we did anyway. But how do we write out the type of a function that takes several parameters? Here's a simple function that takes three integers and adds them together:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

The parameters are separated with -> and there's no special distinction between the parameters and the return type. The return type is the last item in the declaration and the parameters are the first three. Later on we'll see why they're all just separated with -> instead of having some more explicit distinction between the return types and the parameters like `Int, Int, Int -> Int` or something.

If you want to give your function a type declaration but are unsure as to what it should be, you can always just write the function without it and then check it with `:t`. Functions are expressions too, so `:t` works on them without a problem.

Here's an overview of some common types.

`Int` stands for integer. It's used for whole numbers. `7` can be an `Int` but `7.2` cannot. `Int` is bounded, which means that it has a minimum and a maximum value. Usually on 32-bit machines the maximum possible `Int` is 2147483647 and the minimum is -2147483648.

`Integer` stands for, er ... also integer. The main difference is that it's not bounded so it can be used to represent really really big numbers. I mean like really big. `Int`, however, is more efficient.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

Float is a real floating point with single precision.

```
circumference :: Float -&gt; Float
circumference r = 2 * pi * r
```

```
ghci&gt; circumference 4.0
25.132742
```

Double is a real floating point with double the precision!

```
circumference' :: Double -&gt; Double
circumference' r = 2 * pi * r
```

```
ghci&gt; circumference' 4.0
25.132741228718345
```

Bool is a boolean type. It can have only two values: True and False.

Char represents a character. It's denoted by single quotes. A list of characters is a string.

Tuples are types but they are dependent on their length as well as the types of their components, so there is theoretically an infinite number of tuple types, which is too many to cover in this tutorial. Note that the empty tuple () is also a type which can only have a single value: ()

3.2   Type variables

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> :t head
head :: [a] -> a
```

Hmmm! 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.

3.3   Typeclasses 101

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.

What's the type signature of the `==` function?

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

> **Note**: the equality operator, `==` is a function. So are `+`, `*`, `-`, `/` and
> pretty much all operators. If a function is comprised only of special
> characters, it's considered an infix function by default. If we want
> to examine its type, pass it to another function or call it as a prefix
> function, we have to surround it in parentheses.

Interesting. We see a new thing here, the `=>` symbol. Everything before
the `=>` symbol is called a **class constraint**. We can read the previous type
declaration like this: the equality function takes any two values that are of
the same type and returns a `Bool`. The type of those two values must be a
member of the `Eq` class (this was the class constraint).

The `Eq` typeclass provides an interface for testing for equality. Any type
where it makes sense to test for equality between two values of that type
should be a member of the `Eq` class. All standard Haskell types except for IO
(the type for dealing with input and output) and functions are a part of the `Eq`
typeclass.

The `elem` function has a type of `(Eq a) =>a -> [a] -> Bool` because it uses `==`
over a list to check whether some value we're looking for is in it.

Some basic typeclasses:

$\boxed{\texttt{Eq}}$ is used for types that support equality testing. The functions its
members implement are `==` and `/=`. So if there's an `Eq` class constraint for a
type variable in a function, it uses `==` or `/=` somewhere inside its definition.
All the types we mentioned previously except for functions are part of `Eq`, so
they can be tested for equality.

```
ghci> 5 == 5
True
```

```
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

Ord is for types that have an ordering.

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

All the types we covered so far except for functions are part of Ord. Ord covers all the standard comparing functions such as >, <, >= and <=. The compare function takes two Ord members of the same type and returns an ordering. Ordering is a type that can be GT, LT or EQ, meaning *greater than*, *lesser than* and *equal*, respectively.

To be a member of Ord, a type must first have membership in the prestigious and exclusive Eq club.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Members of Show can be presented as strings. All types covered so far except for functions are a part of Show. The most used function that deals with the Show typeclass is show. It takes a value whose type is a member of Show and presents is to us as a string.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Read is sort of the opposite typeclass of Show. The read function takes a string and returns a type which is a member of Read.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

So far so good. Again, all types covered so far are in this typeclass. But what happens if we try to do just read "4"?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable 'a' in the constraint:
```

```
     'Read a' arising from a use of 'read' at <interactive>:1:0-7
   Probable fix: add a type signature that fixes these type variable(s)
```

What GHCI is telling us here is that it doesn't know what we want in
return. Notice that in the previous uses of `read` we did something with the
result afterwards. That way, GHCI could infer what kind of result we wanted
out of our `read`. If we used it as a boolean, it knew it had to return a `Bool`. But
now, it knows we want some type that is part of the `Read` class, it just doesn't
know which one. Let's take a look at the type signature of `read`.

```
ghci> :t read
read :: (Read a) => String -> a
```

See? It returns a type that's part of `Read` but if we don't try to use it in
some way later, it has no way of knowing which type. That's why we can
use explicit **type annotations**. Type annotations are a way of explicitly saying
what the type of an expression should be. We do that by adding `::` at the end
of the expression and then specifying a type. Observe:

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

Most expressions are such that the compiler can infer what their type
is by itself. But sometimes, the compiler doesn't know whether to return a
value of type `Int` or `Float` for an expression like `read "5"`. To see what the
type is, Haskell would have to actually evaluate `read "5"`. But since Haskell
is a statically typed language, it has to know all the types before the code
is compiled (or in the case of GHCI, evaluated). So we have to tell Haskell:
"Hey, this expression should have this type, in case you don't know!".

   `Enum` members are sequentially ordered types — they can be enumerated.
The main advantage of the `Enum` typeclass is that we can use its types in list
ranges. They also have defined successors and predecesors, which you can
get with the `succ` and `pred` functions. Types in this class: `()`, `Bool`, `Char`, `Ordering`,
`Int`, `Integer`, `Float` and `Double`.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

   `Bounded` members have an upper and a lower bound.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
```

```
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` and `maxBound` are interesting because they have a type of `(Bounded a) =>a`. In a sense they are polymorphic constants.

All tuples are also part of `Bounded` if the components are also in it.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

`Num` is a numeric typeclass. Its members have the property of being able to act like numbers. Let's examine the type of a number.

```
ghci> :t 20
20 :: (Num t) => t
```

It appears that whole numbers are also polymorphic constants. They can act like any type that's a member of the `Num` typeclass.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Those are types that are in the `Num` typeclass. If we examine the type of `*`, we'll see that it accepts all numbers.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

It takes two numbers of the same type and returns a number of that type. That's why `(5 :: Int) * (6 :: Integer)` will result in a type error whereas `5 * (6 :: Integer)` will work just fine and produce an `Integer` because `5` can act like an `Integer` or an `Int`.

To join `Num`, a type must already be friends with `Show` and `Eq`.

`Integral` is also a numeric typeclass. `Num` includes all numbers, including real numbers and integral numbers, `Integral` includes only integral (whole) numbers. In this typeclass are `Int` and `Integer`.

`Floating` includes only floating point numbers, so `Float` and `Double`.

A very useful function for dealing with numbers is `fromIntegral`. It has a type declaration of `fromIntegral :: (Num b, Integral a) =>a -> b`. From its type signature we see that it takes an integral number and turns it into a more general number. That's useful when you want integral and floating point types to work together nicely. For instance, the `length` function has a type declaration of `length :: [a] -> Int` instead of having a more general type of `(Num b) =>length :: [a] -> b`. I think that's there for historical reasons or something, although in my opinion, it's pretty stupid. Anyway, if we try to get a

length of a list and then add it to `3.2`, we'll get an error because we tried to
add together an `Int` and a floating point number. So to get around this, we do
`fromIntegral (length [1,2,3,4]) + 3.2` and it all works out.

Notice that `fromIntegral` has several class constraints in its type signature.
That's completely valid and as you can see, the class constraints are separated
by commas inside the parentheses.

Chapter 4

Syntax in Functions

4.1 Pattern matching

This chapter will cover some of Haskell's cool syntactic constructs and we'll start with pattern matching. Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.

When defining functions, you can define separate function bodies for different patterns. This leads to really neat code that's simple and readable. You can pattern match on any data type — numbers, characters, lists, tuples, etc. Let's make a really trivial function that checks if the number we supplied to it is a seven or not.

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

When you call `lucky`, the patterns will be checked from top to bottom and when it conforms to a pattern, the corresponding function body will be used. The only way a number can conform to the first pattern here is if it is 7. If it's not, it falls through to the second pattern, which matches anything and binds it to `x`. This function could have also been implemented by using an if statement. But what if we wanted a function that says the numbers from 1 to 5 and says `"Not between 1 and 5"` for any other number? Without pattern matching, we'd have to make a pretty convoluted if then else tree. However, with it:

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Note that if we moved the last pattern (the catch-all one) to the top, it would always say `"Not between 1 and 5"`, because it would catch all the numbers and they wouldn't have a chance to fall through and be checked for any other

patterns.

Remember the factorial function we implemented previously? We defined
the factorial of a number `n` as `product [1..n]`. We can also define a factorial
function *recursively*, the way it is usually defined in mathematics. We start
by saying that the factorial of 0 is 1. Then we state that the factorial of any
positive integer is that integer multiplied by the factorial of its predecessor.
Here's how that looks like translated in Haskell terms.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

This is the first time we've defined a function recursively. Recursion is
important in Haskell and we'll take a closer look at it later. But in a nut-
shell, this is what happens if we try to get the the factorial of, say, 3. It
tries to compute `3 * factorial 2`. The factorial of 2 is `2 * factorial 1`, so for
now we have `3 * (2 * factorial 1)`. `factorial 1` is `1 * factorial 0`, so we have
`3 * (2 * (1 * factorial 0))`. Now here comes the trick — we've defined the
factorial of 0 to be just 1 and because it encounters that pattern before the catch-
all one, it just returns 1. So the final result is equivalent to `3 * (2 * (1 * 1))`.
Had we written the second pattern on top of the first one, it would catch all
numbers, including 0 and our calculation would never terminate. That's why
order is important when specifying patterns and it's always best to specify
the most specific ones first and then the more general ones later.

Pattern matching can also fail. If we define a function like this:

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

and then try to call it with an input that we didn't expect, this is what
happens:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function
charName
```

It complains that we have non-exhaustive patterns, and rightfully so. When
making patterns, we should always include a catch-all pattern so that our
program doesn't crash if we get some unexpected input.

Pattern matching can also be used on tuples. What if we wanted to make
a function that takes two vectors in a 2D space (that are in the form of pairs)
adds them together. To add together two vectors, we add their x components
separately and then their y components separately. Here's how we would
have done it if we didn't know about pattern matching:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Well, that works, but there's a better way to do it. Let's modify the function so that it uses pattern matching.

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

There we go! Much better. Note that this is already a catch-all pattern. The type of `addVectors` (in both cases) is `addVectors :: (Num a) =>(a, a) -> (a, a) - > (a, a)`, so we are guaranteed to get two pairs as parameters.

`fst` and `snd` extract the components of pairs. But what about triples? Well, there are no provided functions that do that but we can make our own.

```
first :: (a, b, c) -> a
first (x, _, _) = x
second :: (a, b, c) -> b
second (_, y, _) = y
third :: (a, b, c) -> c
third (_, _, z) = z
```

The `_` means the same thing as it does in list comprehensions. It means that we really don't care what that part is, so we just write a `_`.

Which reminds me, you can also pattern match in list comprehensions. Check this out:

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Should a pattern match fail, it will just move on to the next element.

Lists themselves can also be used in pattern matching. You can match with the empty list `[]` or any pattern that involves `:` and the empty list. But since `[1,2,3]` is just syntactic sugar for `1:2:3:[]`, you can also use the former pattern. A pattern like `x:xs` will bind the head of the list to `x` and the rest of it to `xs`, even if there's only one element so `xs` ends up being an empty list.

> **Note**: The `x:xs` pattern is used a lot, especially with recursive functions. But patterns that have `:` in them only match against lists of length 1 or more.

If you want to bind, say, the first three elements to variables and the rest of the list to another variable, you can use something like `x:y:z:zs`. It will only match against lists that have three elements or more.

Now that we know how to pattern match against list, let's make our own implementation of the `head` function.

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

Checking if it works:

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

Nice! Notice that if you want to bind to several variables (even if one of them is just _ and doesn't actually bind at all), we have to surround them in parentheses. Also notice the `error` function that we used. It takes a string and generates a runtime error, using that string as information about what kind of error occured. It causes the program to crash, so it's not good to use it too much. But calling `head` on an empty list doesn't make sense.

Let's make a trivial function that tells us some of the first elements of the list in (in)convinient English form.

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++
" and " ++ show y
```

This function is safe because it takes care of the empty list, a singleton list, a list with two elements and a list with more than two elements. Note that `(x:[])` and `(x:y:[])` could be rewriten as `[x]` and `[x,y]` (because its syntatic sugar, we don't need the parentheses). We can't rewrite `(x:y:_)` with square brackets because it matches any list of length 2 or more.

We already implemented our own `length` function using list comprehension. Now we'll do it by using pattern matching and a little recursion:

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

This is similar to the factorial function we wrote earlier. First we defined the result of a known input — the empty list. This is also known as the edge condition. Then in the second pattern we take the list apart by splitting it into a head and a tail. We say that the length is equal to 1 plus the length of the tail. We use _ to match the head because we don't actually care what it is. Also note that we've taken care of all possible patterns of a list. The first pattern matches an empty list and the second one matches anything that isn't an empty list.

Let's see what happens if we call `length'` on `"ham"`. First, it will check if it's an empty list. Because it isn't, it falls through to the second pattern. It matches on the second pattern and there it says that the length is `1 + length' "am"`, because we broke it into a head and a tail and discarded the tail. O-kay. The `length'` of `"am"` is, similarly, `1 + length' "m"`. So right now we have `1 + (1 + length' "m")`. `length' "m"` is `1 + length' ""` (could also be written as `1 + length' []`). And we've defined `length' []` to be `0`. So in the end we have `1 + (1 + (1 + 0))`.

Let's implement `sum`. We know that the sum of an empty list is `0`. We write that down as a pattern. And we also know that the sum of a list is the head plus the sum of the rest of the list. So if we write that down, we get:

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

There's also a thing called *as patterns*. Those are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing. You do that by putting a name and an @ in front of a pattern. For instance, the pattern `xs@(x:y:ys)`. This pattern will match exactly the same thing as `x:y:ys` but you can easily get the whole list via `xs` instead of repeating yourself by typing out `x:y:ys` in the function body again. Here's a quick and dirty example:

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci&gt; capital "Dracula"
"The first letter of Dracula is D"
```

Normally we use as patterns to avoid repeating ourselves when matching against a bigger pattern when we have to use the whole thing again in the function body.

One more thing — you can't use `++` in pattern matches. If you tried to pattern match against (`xs ++ ys`), what would be in the first and what would be in the second list? It doesn't make much sense. It would make sense to match stuff against (`xs ++ [x,y,z]`) or just (`xs ++ [x]`), but because of the nature of lists, you can't do that.

4.2 Guards, guards!

Whereas patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether some property of a value (or several of them) are true or false. That sounds a lot like an if statement and it's very similar. The thing is that guards are a lot more readable when you have several conditions and they play really nicely with patterns.

Instead of explaining their syntax, let's just dive in and make a function using guards. We're going to make a simple function that berates you differently depending on your BMI (body mass index). Your BMI equals your weight divided by your height squared. If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25 then you're considered normal. 25 to 30 is overweight and more than 30 is obese. So here's the function (we won't be calculating it right now, this function just gets a BMI and tells you off)

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Guards are indicated by pipes that follow a function's name and its parameters. Usually, they're indented a bit to the right and lined up. A guard is basically a boolean expression. If it evaluates to `True`, then the corresponding function body is used. If it evaluates to `False`, checking drops through to the

next guard and so on. If we call this function with 24.3, it will first check if
that's smaller than or equal to 18.5. Because it isn't, it falls through to the next
guard. The check is carried out with the second guard and because 24.3 is
less than 25.0, the second string is returned.

This is very reminiscent of a big if else tree in imperative languages, only
this is far better and more readable. While big if else trees are usually frowned
upon, sometimes a problem is defined in such a discrete way that you can't
get around them. Guards are a very nice alternative for this.

Many times, the last guard is otherwise. otherwise is defined simply as
otherwise = True and catches everything. This is very similar to patterns, only
they check if the input satisfies a pattern but guards check for boolean con-
ditions. If all the guards of a function evaluate to False (and we haven't
provided an otherwise catch-all guard), evaluation falls through to the next
pattern. That's how patterns and guards play nicely together. If no suitable
guards or patterns are found, an error is thrown.

Of course we can use guards with functions that take as many parameters
as we want. Instead of having the user calculate his own BMI before calling
the function, let's modify this function so that it takes a height and weight
and calculates it for us.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet
you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Let's see if I'm fat . . .

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

Yay! I'm not fat! But Haskell just called me ugly. Whatever!

Note that there's no = right after the function name and its parameters,
before the first guard. Many newbies get syntax errors because they sometimes
put it there.

Another very simple example: let's implement our own max function. If
you remember, it takes two things that can be compared and returns the larger
of them.

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b
```

Guards can also be written inline, although I'd advise against that because
it's less readable, even for very short functions. But to demonstrate, we could
write max' like this:

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Ugh! Not very readable at all! Moving on: let's implement our own `compare` by using guards.

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b      = GT
    | a == b     = EQ
    | otherwise = LT
```

4.3 Where!?

In the previous section, we defined a BMI calculator function and berator like this:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet
you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Notice that we repeat ourselves here three times. We repeat ourselves three times. Repeating yourself (three times) while programming is about as desirable as getting kicked inna head. Since we repeat the same expression three times, it would be ideal if we could calculate it once, bind it to a name and then use that name instead of the expression. Well, we can modify our function like this:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

We put the keyword `where` after the guards (usually it's best to indent it as much as the pipes are indented) and then we define several names or functions. These names are visible across the guards and give us the advantage of not having to repeat ourselves. If we decide that we want to calculate BMI a bit differently, we only have to change it once. It also improves readability by giving names to things and can make our programs faster since stuff like our `bmi` variable here is calculated only once. We could go a bit overboard and present our function like this:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

The names we define in the where section of a function are only visible to that function, so we don't have to worry about them polluting the namespace

of other functions. Notice that all the names are aligned at a single column. If
we don't align them nice and proper, Haskell gets confused because then it
doesn't know they're all part of the same block.

where bindings aren't shared across function bodies of different patterns. If
you want several patterns of one function to access some shared name, you
have to define it globally.

You can also use where bindings to **pattern match**! We could have rewritten
the where section of our previous function as:

```
   ...
   where bmi = weight / height ^ 2
         (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Let's make another fairly trivial function where we get a first and a last
name and give someone back their initials.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

We could have done this pattern matching directly in the function's pa-
rameters (it would have been shorter and clearer actually) but this just goes to
show that it's possible to do it in where bindings as well.

Just like we've defined constants in where blocks, you can also define
functions. Staying true to our healthy programming theme, let's make a
function that takes a list of weight-height pairs and returns a list of BMIs.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

And that's all there is to it! The reason we had to introduce `bmi` as a
function in this example is because we can't just calculate one BMI from the
function's parameters. We have to examine the list passed to the function and
there's a different BMI for every pair in there.

where bindings can also be nested. It's a common idiom to make a function
and define some helper function in its *where* clause and then to give those
functions helper functions as well, each with its own *where* clause.

4.4　Let it be

Very similar to where bindings are let bindings. Where bindings are a syntactic
construct that let you bind to variables at the end of a function and the whole
function can see them, including all the guards. Let bindings let you bind to
variables anywhere and are expressions themselves, but are very local, so they
don't span across guards. Just like any construct in Haskell that is used to
bind values to names, let bindings can be used for pattern matching. Let's
see them in action! This is how we could define a function that gives us a
cylinder's surface area based on its height and radius:

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

The form is `let <bindings> in <expression>`. The names that you define in the *let* part are accessible to the expression after the *in* part. As you can see, we could have also defined this with a *where* binding. Notice that the names are also aligned in a single column. So what's the difference between the two? For now it just seems that *let* puts the bindings first and the expression that uses them later whereas *where* is the other way around.

The difference is that *let* bindings are expressions themselves. *where* bindings are just syntactic constructs. Remember when we did the if statement and it was explained that an if else statement is an expression and you can cram it in almost anywhere?

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

You can also do that with let bindings.

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

They can also be used to introduce functions in a local scope:

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

If we want to bind to several variables inline, we obviously can't align them at columns. That's why we can separate them with semicolons.

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!"
in foo ++ bar)
(6000000,"Hey there!")
```

You don't have to put a semicolon after the last binding but you can if you want. Like we said before, you can pattern match with *let* bindings. They're very useful for quickly dismantling a tuple into components and binding them to names and such.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

You can also put *let* bindings inside list comprehensions. Let's rewrite our previous example of calculating lists of weight-height pairs to use a *let* inside a list comprehension instead of defining an auxiliary function with a *where*.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

We include a *let* inside a list comprehension much like we would a predicate, only it doesn't filter the list, it only binds to names. The names defined in a *let* inside a list comprehension are visible to the output function (the part before the |) and all predicates and sections that come after of the binding. So we could make our function return only the BMIs of fat people:

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

We can't use the `bmi` name in the `(w, h) <- xs` part because it's defined prior to the *let* binding.

We omitted the *in* part of the *let* binding when we used them in list comprehensions because the visibility of the names is already predefined there. However, we could use a *let in* binding in a predicate and the names defined would only be visible to that predicate. The *in* part can also omitted when defining functions and constants directly in GHCI. If we do that, then the names will be visible throughout the entire interactive session.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: 'boot'
```

If *let* bindings are so cool, why not use them all the time instead of *where* bindings, you ask? Well, since *let* bindings are expressions and are fairly local in their scope, they can't be used across guards. Some people prefer *where* bindings because the names come after the function they're being used in. That way, the function body is closer to its name and type declaration and some that's more readable.

4.5   Case expressions

Many imperative languages (C, C++, Java, etc.) have case syntax and if you've ever programmed in them, you probably know what it's about. It's about taking a variable and then executing blocks of code for specific values of that variable and then maybe including a catch-all block of code in case the variable has some value for which we didn't set up a case.

Haskell takes that concept and one-ups it. Like the name implies, case expressions are, well, expressions, much like if else expressions and *let* bindings. Not only can we evaluate expressions based on the possible cases of the value of a variable, we can also do pattern matching. Hmmm, taking a variable, pattern matching it, evaluating pieces of code based on its value, where have we heard this before? Oh yeah, pattern matching on parameters in function definitions! Well, that's actually just syntactic sugar for case expressions. These two pieces of code do the same thing and are interchangeable:

```
head' :: [a] -&gt; a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

As you can see, the syntax for case expressions is pretty simple:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` is matched against the patterns. The pattern matching action is the same as expected: the first pattern that matches the expression is used. If it falls through the whole case expression and no suitable pattern is found, a runtime error occurs.

Whereas pattern matching on function parameters can only be done when defining functions, case expressions can be used pretty much anywhere. For instance:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

They're are useful for pattern matching against something in the middle of an expression. Because pattern matching in function definitions is syntactic sugar for case expressions, we could have also defined this like so:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Chapter 5

Recursion

5.1 Hello recursion!

We mention recursion briefly in the previous chapter. In this chapter, we'll take a closer look at recursion, why it's important to Haskell and how we can work out very concise and elegant solutions to problems by thinking recursively.

If you still don't know what recursion is, read this sentence. Haha! Just kidding! Recursion is actually a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively. For instance, the fibonacci sequence is defined recursively. First, we define the first two fibonacci numbers non-recursively. We say that *F(0) = 0* and *F(1) = 1*, meaning that the 0th and 1st fibonacci numbers are 0 and 1, respectively. Then we say that for any other natural number, that fibonacci number is the sum of the previous two fibonacci numbers. So *F(n) = F(n-1) + F(n-2)*. That way, *F(3)* is *F(2) + F(1)*, which is *(F(1) + F(0)) + F(1)*. Because we've now come down to only non-recursively defined fibonacci numbers, we can safely say that *F(3)* is 2. Having an element or two in a recursion definition defined non-recursively (like *F(0)* and *F(1)* here) is also called the **edge condition** and is important if you want your recursive function to terminate. If we hadn't defined *F(0)* and *F(1)* non recursively, you'd never get a solution any number because you'd reach 0 and then you'd go into negative numbers. All of a sudden, you'd be saying that *F(-2000)* is *F(-2001) + F(-2002)* and there still wouldn't be an end in sight!

Recursion is important to Haskell because unlike imperative languages, you do computations in Haskell by declaring what something *is* instead of declaring *how* you get it. That's why there are no while loops or for loops in Haskell and instead we many times have to use recursion to declare what something is.

5.2   Maximum awesome

The `maximum` function takes a list of things that can be ordered (e.g. instances of the `Ord` typeclass) and returns the biggest of them. Think about how you'd implement that in an imperative fashion. You'd probably set up a variable to hold the maximum value so far and then you'd loop through the elements of a list and if an element is bigger than then the current maximum value, you'd replace it with that element. The maximum value that remains at the end is the result. Whew! That's quite a lot of words to describe such a simple algorithm!

Now let's see how we'd define it recursively. We could first set up an edge condition and say that the maximum of a singleton list is equal to the only element in it. Then we can say that the maximum of a longer list is the head if the head is bigger than the maximum of the tail. If the maximum of the tail is bigger, well, then it's the maximum of the tail. That's it! Now let's implement that in Haskell.

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

As you can see, pattern matching goes great with recursion! Most imperative languages don't have pattern matching so you have to make a lot of if else statements to test for edge conditions. Here, we simply put them out as patterns. So the first edge condition says that if the list is empty, crash! Makes sense because what's the maximum of an empty list? I don't know. The second pattern also lays out an edge condition. It says that if it's the singleton list, just give back the only element.

Now the third pattern is where the action happens. We use pattern matching to split a list into a head and a tail. This is a very common idiom when doing recursion with lists, so get used to it. We use a *where* binding to define `maxTail` as the maximum of the rest of the list. Then we check if the head is greater than the maximum of the rest of the list. If it is, we return the head. Otherwise, we return the maximum of the rest of the list.

Let's take an example list of numbers and check out how this would work on them: `[2,5,1]`. If we call `maximum'` on that, the first two patterns won't match. The third one will and the list is split into `2` and `[5,1]`. The *where* clause wants to know the maximum of `[5,1]`, so we follow that route. It matches the third pattern again and `[5,1]` is split into `5` and `[1]`. Again, the `where` clause wants to know the maximum of `[1]`. Because that's the edge condition, it returns `1`. Finally! So going up one step, comparing `5` to the maximum of `[1]` (which is `1`), we obviously get back `5`. So now we know that the maximum of `[5,1]` is `5`. We go up one step again where we had `2` and `[5,1]`. Comparing `2` with the maximum of `[5,1]`, which is `5`, we choose `5`.

An even clearer way to write this function is to use `max`. If you remember, `max` is a function that takes two numbers and returns the bigger of them. Here's how we could rewrite `maximum'` by using `max`:

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

How's that for elegant! In essence, the maximum of a list is the max of the first element and the maximum of the tail.

5.3 A few more recursive functions

Now that we know how to generally think recursively, let's implement a few functions using recursion. First off, we'll implement `replicate`. `replicate` takes an `Int` and some element and returns a list that has several repetitions of the same element. For instance, `replicate 3 5` returns `[5,5,5]`. Let's think about the edge condition. My guess is that the edge condition is 0 or less. If we try to replicate something zero times, it should return an empty list. Also for negative numbers, because it doesn't really make sense.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

We used guards here instead of patterns because we're testing for a boolean condition. If `n` is less than or equal to 0, return an empty list. Otherwise return a list that has `x` as the first element and then `x` replicated n-1 times as the tail. Eventually, the `(n-1)` part will cause our function to reach the edge condition.

> **Note:** `Num` is not a subclass of `Ord`. That means that what constitutes for a number doesn't really have to adhere to an ordering. So that's why we have to specify both the `Num` and `Ord` class constraints when doing addition or subtraction and also comparison.

Next up, we'll implement `take`. It takes a certain number of elements from a list. For instance, `take 3 [5,4,3,2,1]` will return `[5,4,3]`. If we try to take 0 or less elements from a list, we get an empty list. Also if we try to take anything from an empty list, we get an empty list. Notice that those are two edge conditions right there. So let's write that out:

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

The first pattern specifies that if we try to take a 0 or negative number of elements, we get an empty list. Notice that we're using `_` to match the list because we don't really care what it is in this case. Also notice that we use a guard, but without an `otherwise` part. That means that if `n` turns out to be more than 0, the matching will fall through to the next pattern. The second pattern indicates that if we try to take anything from an empty list, we get an empty list.

The third pattern breaks the list into a head and a tail. And then we state that taking n elements from a list equals a list that has x as the head and then a list that takes n-1 elements from the tail as a tail. Try using a piece of paper to write down how the evaluation would look like if we try to take, say, 3 from [4,3,2,1].

reverse simply reverses a list. Think about the edge condition. What is it? Come on ... it's the empty list! An empty list reversed equals the empty list itself. O-kay. What about the rest of it? Well, you could say that if we split a list to a head and a tail, the reversed list is equal to the reversed tail and then the head at the end.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

There we go!

Because Haskell supports infinite lists, our recursion doesn't really have to have an edge condition. But if it doesn't have it, it will either keep churning at something infinitely or produce an infinite data structure, like an infinite list. The good thing about infinite lists though is that we can cut them where we want. repeat takes an element and returns an infinite list that just has that element. A recursive implementation of that is really easy, watch.

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Calling repeat 3 will give us a list that starts with 3 and then has an infinite amount of 3's as a tail. So calling repeat 3 would evaluate like 3:repeat 3, which is 3:(3:repeat 3), which is 3:(3:(3:repeat 3)), etc. repeat 3 will never finish evaluating, whereas take 5 (repeat 3) will give us a list of five 3's. So essentially it's like doing replicate 5 3.

zip takes two lists and zips them together. zip [1,2,3] [2,3] returns [(1,2),(2,3)], because it truncates the longer list to match the length of the shorter one. How about if we zip something with an empty list? Well, we get an empty list back then. So there's our edge condition. However, zip takes two lists as parameters, so there are actually two edge conditions.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

First two patterns say that if the first list or second list is empty, we get an empty list. The third one says that two lists zipped are equal to pairing up their heads and then tacking on the zipped tails. Zipping [1,2,3] and ['a','b'] will eventually try to zip [3] with []. The edge condition patterns kick in and so the result is (1,'a'):(2,'b'):[], which is exactly the same as [(1,'a'),(2,'b')].

Let's implement one more standard library function — elem. It takes an element and a list and sees if that element is in the list. The edge condition, as is most of the times with lists, is the empty list. We know that an empty list contains no elements, so it certainly doesn't have the droids we're looking for.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Pretty simple and expected. If the head isn't the element then we check the tail. If we reach an empty list, the result is `False`.

5.4   Quick, sort!

We have a list of items that can be sorted. Their type is an instance of the `Ord` typeclass. And now, we want to sort them! There's a very cool algoritm for sorting called quicksort. It's a very clever way of sorting items. While it takes upwards of 10 lines to implement quicksort in imperative languages, the implementation is much shorter and elegant in Haskell. Quicksort has become a sort of poster child for Haskell. Therefore, let's implement it here, even though implementing quicksort in Haskell is considered really cheesy because everyone does it to showcase how elegant Haskell is.

So, the type signature is going to be `quicksort :: (Ord a) =>[a] -> [a]`. No surprises there. The edge condition? Empty list, as is expected. A sorted empty list is an empty list. Now here comes the main algorithm: **a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).** Notice that we said *sorted* two times in this definition, so we'll probably have to make the recursive call twice! Also notice that we defined it using the verb *is* to define the algorithm instead of saying *do this, do that, then do that ....* That's the beauty of functional programming! How are we going to filter the list so that we get only the elements smaller than the head of our list and only elements that are bigger? List comprehensions. So, let's dive in and define this function.

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Let's give it a small test run to see if it appears to behave correctly.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

Booyah! That's what I'm talking about! So if we have, say `[5,1,9,4,6,7,3]` and we want to sort it, this algorithm will first take the head, which is `5` and then put it in the middle of two lists that are smaller and bigger than it. So at one point, you'll have `[1,4,3] ++ [5] ++ [9,6,7]`. We know that once the list is sorted completely, the number `5` will stay in the fourth place since there are 3 numbers lower than it and 3 numbers higher than it. Now, if we sort `[1,4,3]` and `[9,6,7]`, we have a sorted list! We sort the two lists using the same

function. Eventually, we'll break it up so much that we reach empty lists and an empty list is already sorted in a way, by virtue of being empty. Here's an illustration:

An element that is in place and won't move anymore is represented in **orange**. If you read them from left to right, you'll see the sorted list. Although we chose to compare all the elements to the heads, we could have used any element to compare against. In quicksort, an element that you compare against is called a pivot. They're in **green** here. We chose the head because it's easy to get by pattern matching. The elements that are smaller than the pivot are **light green** and elements larger than the pivot are **dark green**. The yellowish gradient thing represents an application of quicksort.

5.5   Thinking recursively

We did quite a bit of recursion so far and as you've probably noticed, there's a pattern here. Usually you define an edge case and then you define a function that does something between some element and the function applied to the rest. It doesn't matter if it's a list, a tree or any other data structure. A sum is the first element of a list plus the sum of the rest of the list. A product of a list is the first element of the list times the product of the rest of the list. The length of a list is one plus the length of the tail of the list. Ekcetera, ekcetera ...

Of course, these also have edge cases. Usually the edge case is some scenario where a recursive application doesn't make sense. When dealing with lists, the edge case is most often the empty list. If you're dealing with trees, the edge case is usually a node that doesn't have any children.

It's similar when you're dealing with numbers recursively. Usually it has to do with some number and the function applied to that number modified. We did the factorial function earlier and it's the product of a number and the factorial of that number minus one. Such a recursive application doesn't make sense with zero, because factorials are defined only for positive integers. Often the edge case value turns out to be an identity. The identity for multiplication is 1 because if you multiply something by 1, you get that something back. Also when doing sums of lists, we define the sum of an empty list as 0 and 0 is the identity for addition. In quicksort, the edge case is the empty list and the identity is also the empty list, because if you add an empty list to a list, you just get the original list back.

So when trying to think of a recursive way to solve a problem, try to think of when a recursive solution doesn't apply and see if you can use that as an edge case, think about identities and think about whether you'll break apart the parameters of the function (for instance, lists are usually broken into a head and a tail via pattern matching) and on which part you'll use the recursive call.

Chapter 6

Higher order functions

Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience. It turns out that if you want to define computations by defining what stuff *is* instead of defining steps that change some state and maybe looping them, higher order functions are indispensable. They're a really powerful way of solving problems and thinking about programs.

6.1   Curried functions

Every function in Haskell officially only takes one parameter. So how is it possible that we defined and used several functions that take more than one parameter so far? Well, it's a clever trick! All the functions that accepted *several parameters* so far have been **curried functions**. What does that mean? You'll understand it best on an example. Let's take our good friend, the `max` function. It looks like it takes two parameters and returns the one that's bigger. Doing `max 4 5` first creates a function that takes a parameter and returns either `4` or that parameter, depending on which is bigger. Then, `5` is applied to that function and that function produces our desired result. That sounds like a mouthful but it's actually a really cool concept. The following two calls are equivalent:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Putting a space between two things is simply ¡b¿function application¡/b¿. The space is sort of like an operator and it has the highest precedence. Let's examine the type of `max`. It's `max :: (Ord a) =>a -> a -> a`. That can also be written as `max :: (Ord a) =>a -> (a -> a)`. That could be read as: `max` takes an `a` and returns (that's the ->) a function that takes an `a` and returns an `a`. That's why the return type and the parameters of functions are all simply separated with arrows.

So how is that beneficial to us? Simply speaking, if we call a function with too few parameters, we get back a **partially applied** function, meaning a function that takes as many parameters as we left out. Using partial application (calling functions with too few parameters, if you will) is a neat way to create functions on the fly so we can pass them to another function or to seed them with some data.

Take a look at this offensively simple function:

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

What really happens when we do `multThree 3 5 9` or `((multThree 3) 5) 9`? First, `3` is applied to `multThree`, because they're separated by a space. That creates a function that takes one parameter and returns a function. So then `5` is applied to that, which creates a function that will take a parameter and multiply it by 15. `9` is applied to that function and the result is 135 or something. Remember that this function's type could also be written as `multThree :: (Num a) =>a -> (a -> (a -> a))`. The thing before the `->` is the parameter that a function takes and the thing after it is what it returns. So our function takes an `a` and returns a function of type `(Num a) =>a -> (a -> a)`. Similarly, this function takes an `a` and returns a function of type `(Num a) =>a -> a`. And this function, finally, just takes an `a` and returns an `a`. Take a look at this:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

By calling functions with too few parameters, so to speak, we're creating new functions on the fly. What if we wanted to create a function that takes a number and compares it to `100`? We could do something like this:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

If we call it with `99`, it returns a `GT`. Simple stuff. Notice that the `x` is on the right hand side on both sides of the equation. Now let's think about what `compare 100` returns. It returns a function that takes a number and compares it with `100`. Wow! Isn't that the function we wanted? We can rewrite this as:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100
```

The type declaration stays the same, because `compare 100` returns a function. Compare has a type of `(Ord a) =>a -> (a -> Ordering)` and calling it with `100` returns a `(Num a, Ord a) =>a -> Ordering`. The additional class constraint sneaks up there because `100` is also part of the `Num` typeclass.

> **Yo!** Make sure you really understand how curried functions and partial application work because they're really important!

Infix functions can also be partially applied by using sections. To section an infix function, simply surround it with parentheses and only supply a parameter on one side. That creates a function that takes one parameter and

then applies it to the side that's missing an operand. An insultingly trivial function:

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

Calling, say, `divideByTen 200` is equivalent to doing `200 / 10`, as is doing `(/10) 200`. A function that checks if a character supplied to it is an uppercase letter:

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

The only special thing about sections is using `-`. From the definition of sections, `(-4)` would result in a function that takes a number and subtracts 4 from it. However, for convenience, `(-4)` means minus four. So if you want to make a function that subtracts 4 from the number it gets as a parameter, partially apply the `subtract` function like so: `(subtract 4)`.

What happens if we try to just do `multThree 3 4` in GHCI instead of binding it to a name with a *let* or passing it to another function?

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (t -> t))
      arising from a use of 'print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a 'do' expression: print it
```

GHCI is telling us that the expression produced a function of type `a -> a` but it doesn't know how to print it to the screen. Functions aren't instances of the `Show` typeclass, so we can't get a neat string representation of a function. When we do, say, `1 + 1` at the GHCI prompt, it first calculates that to `2` and then calls `show` on `2` to get a textual representation of that number. And the textual representation of `2` is just the string `"2"`, which then gets printed to our screen.

6.2   Some higher-orderism is in order

Functions can take functions as parameters and also return functions. To illustrate this, we're going to make a function that takes a function and then applies it twice to something!

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

First of all, notice the type declaration. Before, we didn't need parentheses because `->` is naturally right-associative. However, here, they're mandatory. They indicate that the first parameter is a function that takes something and returns that same thing. The second parameter is something of that type also and the return value is also of the same type. We could read this type declaration in the curried way, but to save ourselves a headache, we'll just say that this function takes two parameters and returns one thing. The first parameter is a function (of type `a -> a`) and the second is that same `a`. The function can also be `Int -> Int` or `String -> String` or whatever. But then, the second parameter to also has to be of that type.

Note: From now on, we'll say that functions take several parameters despite each function actually taking only one parameter and returning partially applied functions until we reach a function that returns a solid value. So for simplicity's sake, we'll say that `a -> a -> a` takes two parameters, even though we know what's really going on under the hood.

The body of the function is pretty simple. We just use the parameter `f` as a function, applying `x` to it by separating them with a space and then applying the result to `f` again. Anyway, playing around with the function:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

The awesomeness and usefulness of partial application is evident. If our function requires us to pass it a function that takes only one parameter, we can just partially apply a function to the point where it takes only one parameter and then pass it.

Now we're going to use higher order programming to implement a really useful function that's in the standard library. It's called `zipWith`. It takes a function and two lists as parameters and then joins the two lists by applying the function between corresponding elements. Here's how we'll implement it:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Look at the type declaration. The first parameter is a function that takes two things and produces a third thing. They don't have to be of the same type, but they can. The second and third parameter are lists. The result is also a list. The first has to be a list of `a`'s, because the joining function takes `a`'s as its first argument. The second has to be a list of `b`'s, because the second parameter of the joining function is of type `b`. The result is a list of `c`'s. If the type declaration of a function says it accepts an `a -> b -> c` function as a parameter, it will also accept an `a -> a -> a` function, but not the other way around! Remember that when you're making functions, especially higher order ones, and you're unsure of the type, you can just try omitting the type declaration and then checking what Haskell infers it to be by using `:t`.

The action in the function is pretty similar to the normal `zip`. The edge conditions are the same, only there's an extra argument, the joining function, but that argument doesn't matter in the edge conditions, so we just use a `_` for it. And function body at the last pattern is also similar to `zip`, only it doesn't do `(x,y)`, but `f x y`. A single higher order function can be used for a multitude of different tasks if it's general enough. Here's a little demonstration of all the different things our `zipWith'` function can do:

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers",
"aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]]
[[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

As you can see, a single higher order function can be used in very versatile ways. Imperative programming usually uses stuff like for loops, while loops, setting something to a variable, checking its state, etc. to achieve some behavior and then wrap it around an interface, like a function. Functional programming uses higher order functions to abstract away common patterns, like examining two lists in pairs and doing something with those pairs or getting a set of solutions and eliminating the ones you don't need.

We'll implement another function that's already in the standard library, called `flip`. Flip simply takes a function and returns a function that is like our original function, only the first two arguments are flipped. We can implement it like so:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

Reading the type declaration, we say that it takes a function that takes an `a` and a `b` and returns a function that takes a `b` and an `a`. But because functions are curried by default, the second pair of parentheses is really unnecessary, because `->` is right associative by default. `(a -> b -> c) -> (b -> a -> c)` is the same as `(a -> b -> c) -> (b -> (a -> c))`, which is the same as `(a -> b -> c) -> b -> a -> c`. We wrote that `g x y = f y x`. If that's true, then `f y x = g x y` must also hold, right? Keeping that in mind, we can define this function in an even simpler manner.

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

Here, we take advantage of the fact that functions are curried. When we call `flip' f` without the parameters `y` and `x`, it will return an `f` that takes those two parameters but calls them flipped. Even though flipped functions are usually passed to other functions, we can take advantage of currying when making higher-order functions by thinking ahead and writing what their end result would be if they were called fully applied.

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

6.3   Maps and filters

`map` takes a function and a list and applies that function to every element in
the list, producing a new list. Let's see what its type signature is and how it's
defined.

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

The type signature says that it takes a function that takes an `a` and returns
a `b`, a list of `a`'s and returns a list of `b`'s. It's interesting that just by looking at
a function's type signature, you can sometimes tell what it does. `map` is one of
those really versatile higher-order functions that can be used in millions of
different ways. Here it is in action:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

You've probably noticed that each of these could be achieved with a list com-
prehension. `map (+3) [1,5,3,1,6]` is the same as writing `[x+3 | x <- [1,5,3,1,6]]`.
However, using `map` is much more readable for cases where you only apply
some function to the elements of a list, especially once you're dealing with
maps of maps and then the whole thing with a lot of brackets can get a bit
messy.

`filter` is a function that takes a predicate (a predicate is a function that
tells whether something is true or not, so in our case, a function that returns
a boolean value) and a list and then returns the list of elements that satisfy
the predicate. The type signature and implementation go like this:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Pretty simple stuff. If `p x` evaluates to `True`, the element gets included in
the new list. If it doesn't, it stays out. Some usage examples:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull
[[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
```

`"GAYBALLS"`

All of this could also be achived with list comprehensions by the use of predicates. There's no set rule for when to use `map` and `filter` versus using list comprehension, you just have to decide what's more readable depending on the code and the context. The `filter` equivalent of applying several predicates in a list comprehension is either filtering something several times or joining the predicates with the logical `&&` function.

Remember our quicksort function from the previous chapter? We used list comprehensions to filter out the list elements that are smaller than (or equal to) and larger than the pivot. We can achieve the same functionality in a more readable way by using `filter`:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<=x) xs)
        biggerSorted = quicksort (filter (>x) xs)
    in  smallerSorted ++ [x] ++ biggerSorted
```

Mapping and filtering is the bread and butter of every functional programmer's toolbox. Uh. It doesn't matter if you do it with the `map` and `filter` functions or list comprehensions. Recall how we solved the problem of finding right triangles with a certain circumference. With imperative programming, we would have solved it by nesting three loops and then testing if the current combination satisfies a right triangle and if it has the right perimeter. If that's the case, we would have printed it out to the screen or something. In functional programming, that pattern is achieved with mapping and filtering. You make a function that takes a value and produces some result. We map that function over a list of values and then we filter the resulting list out for the results that satisfy our search. Thanks to Haskell's laziness, even if you map something over a list several times and filter it several times, it will only pass over the list once.

Let's **find the largest number under 100,000 that's divisible by 3829**. To do that, we'll just filter a set of possibilities in which we know the solution lies.

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3892 == 0
```

We first make a list of all numbers lower than 100,000, descending. Then we filter it by our predicate and because the numbers are sorted in a descending manner, the largest number that satisfies our predicate is the first element of the filtered list. We didn't even need to use a finite list for our starting set. That's laziness in action again. Because we only end up using the head of the filtered list, it doesn't matter if the filtered list is finite or infinite. The evaluation stops when the first adequate solution is found.

Next up, we're going to **find the sum of all odd squares that are smaller than 10,000**. But first, because we'll be using it in our solution, we're going to introduce the ¡`takeWhile` function. It takes a predicate and a list and then goes

from the beginning of the list and returns its elements while the predicate holds
true. Once an element is found for which the predicate doesn't hold, it stops.
If we wanted to get the first word of the string `"elephants know how to party"`,
we could do `takeWhile (/=' ') "elephants know how to party"` and it would return
`"elephants"`. Okay. The sum of all odd squares that are smaller than 10,000.
First, we'll begin by mapping the `(^2)` function to the infinite list `[1..]`. Then
we filter them so we only get the odd ones. And then, we'll take elements
from that list while they are smaller than 10,000. Finally, we'll get the sum of
that list. We don't even have to define a function for that, we can do it in one
line in GHCI:

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

Awesome! We start with some initial data (the infinite list of all natural
numbers) and then we map over it, filter it and cut it until it suits our needs
and then we just sum it up. We could have also written this using list
comprehensions:

```
ghci> sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])
166650
```

It's a matter of taste as to which one you find prettier. Again, Haskell's
property of laziness is what makes this possible. We can map over and filter
an infinite list, because it won't actually map and filter it right away, it'll delay
those actions. Only when we force Haskell to show us the sum does the `sum`
function say to the `takeWhile` that it needs those numbers. `takeWhile` forces the
filtering and mapping to occur, but only until a number greater than or equal
to 10,000 is encountered.

For our next problem, we'll be dealing with Collatz sequences. We take
a natural number. If that number is even, we divide it by two. If it's odd,
we multiply it by 3 and then add 1 to that. We take the resulting number
and apply the same thing to it, which produces a new number and so on. In
essence, we get a chain of numbers. It is thought that for all starting numbers,
the chains finish at the number 1. So if we take the starting number 13, we
get this sequence: *13, 40, 20, 10, 5, 16, 8, 4, 2, 1*. 13*3 + 1 equals 40. 40 divided
by 2 is 20, etc. We see that the chain has 10 terms.

Now what we want to know is this: **for all starting numbers between
1 and 100, how many chains have a length greater than 15?** First off, we'll
write a function that produces a chain:

```
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

Because the chains end at 1, that's the edge case. This is a pretty standard
recursive function.

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
```

```
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Yay! It seems to be working correctly. And now, the function that tells us the answer to our question:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

We map the `chain` function to `[1..100]` to get a list of chains, which are themselves represented as lists. Then, we filter them by a predicate that just checks whether a list's length is longer than 15. Once we've done the filtering, we see how many chains are left in the resulting list.

> **Note:** This function has a type of `numLongChains :: Int` because `length` returns an `Int` instead of a `Num a` for historical reasons. If we wanted to return a more general `Num a`, we could have used `fromIntegral` on the resulting length.

Using `map`, we can also do stuff like `map (*) [0..]`, if not for any other reason than to illustrate how currying works and how (partially applied) functions are real values that you can pass around to other functions or put into lists (you just can't turn them to strings). So far, we've only mapped functions that take one parameter over lists, like `map (*2) [0..]` to get a list of type `(Num a) =>[a]`, but we can also do `map (*) [0..]` without a problem. What happens here is that the number in the list is applied to the function `*`, which has a type of `(Num a) =>a -> a -> a`. Applying only one parameter to a function that takes two parameters returns a function that takes one parameter. If we map `*` over the list `[0..]`, we get back a list of functions that only take one parameter, so `(Num a) =>[a -> a]`. `map (*) [0..]` produces a list like the one we'd get by writing `[(*0),(*1),(*2),(*3),(*4),(*5)...`

```
ghci> let listOfFuns = map (*) [0..]
ghci> (liftOfFuns !! 4) 5
20
```

Getting the element with the index `4` from our list returns a function that's equivalent to `(*4)`. And then, we just apply `5` to that function. So that's like writing `(* 4) 5` or just `4 * 5`.

6.4 Lambdas

Lambdas are basically anonymous functions that are used because we need some functions only once. Normally, we make a lambda with the sole purpose of passing it to a higher-order function. To make a lambda, we write a \ (because it kind of looks like the greek letter lambda if you squint hard enough) and then we write the parameters, separated by spaces. After that comes a -> and then the function body. We usually surround them by parentheses, because otherwise they extend all the way to the right.

If you look about 5 inches up, you'll see that we used a *where* binding in our `numLongChains` function to make the `isLong` function for the sole purpose of passing it to `filter`. Well, instead of doing that, we can use a lambda:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```

Lambdas are expressions, that's why we can just pass them like that. The expression `(\xs -> length xs > 15)` returns a function that tells us whether the length of the list passed to it is greater than 15.

People who are not well acquainted with how currying and partial application works often use lambdas where they don't need to. For instance, the expressions `map (+3) [1,6,3,2]` and `map (\x -> x + 3) [1,6,3,2]` are equivalent since both `(+3)` and `(\x -> x + 3)` are functions that take a number and add 3 to it. Needless to say, making a lambda in this case is stupid since using partial application is much more readable.

Like normal functions, lambdas can take any number of parameters:

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

And like normal functions, you can pattern match in lambdas. The only difference is that you can't define several patterns for one parameter, like making a `[]` and a `(x:xs)` pattern for the same parameter and then having values fall through. If a pattern matching fails in a lambda, a runtime error occurs, so be careful when pattern matching in lambdas!

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Lambdas are normally surrounded by parentheses unless we mean for them to extend all the way to the right. Here's something interesting: due to the way functions are curried by default, these two are equivalent:

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

If we define a function like this, it's obvious why the type declaration is what it is. There are three ->'s in both the type declaration and the equation. But of course, the first way to write functions is far more readable, the second one is pretty much a gimmick to illustrate currying. However, there are times when using this notation is cool. I think that the `flip` function is the most readable when defined like so:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

Even though that's the same as writing `flip' f x y = f y x`, we make it obvious that this will be used for producing a new function most of the time. The most common use case with `flip` is calling it with just the function parameter and then passing the resulting function on to a map or a filter. So use lambdas in this way when you want to make it explicit that your function is mainly meant to be partially applied and passed on to a function as a parameter.

6.5 Only folds and horses

Back when we were dealing with recursion, we noticed a theme throughout many of the recursive functions that operated on lists. Usually, we'd have an edge case for the empty list. We'd introduce the `x:xs` pattern and then we'd do some action that involves a single element and the rest of the list. It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called folds. They're sort of like the `map` function, only they reduce the list to some single value.

A fold takes a binary function, a starting value (I like to call it the accumulator) and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first (or last) element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

First let's take a look at the `foldl` function, also called the left fold. It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc.

Let's implement `sum` again, only this time, we'll use a fold instead of explicit recursion.

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Testing, one two three:

```
ghci> sum' [3,5,2,1]
11
```

Let's take an in-depth look into how this fold happens. `\acc x -> acc + x` is the binary function. `0` is the starting value and `xs` is the list to be folded up. Now first, `0` is used as the `acc` parameter to the binary function and `3` is used as the `x` (or the current element) parameter. `0 + 3` produces a `3` and it becomes the new accumulator value, so to speak. Next up, `3` is used as the accumulator value and `5` as the current element and `8` becomes the new accumulator value. Moving forward, `8` is the accumulator value, `2` is the current element, the new accumulator value is `10`. Finally, that `10` is used as the accumulator value and `1` as the current element, producing an `11`. Congratulations, you've done a fold!

This professional diagram on the left illustrates how a fold happens, step by step (day by day!). The greenish brown number is the accumulator value. You can see how the list is sort of consumed up from the left side by the accumulator. Om nom nom nom! If we take into account that functions are curried, we can write this implementation ever more succinctly, like so:

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

The lambda function (`\acc x -> acc + x`) is the same as (`+`). We can omit the `xs` as the parameter because calling `foldl (+) 0` will return a function that

takes a list. Generally, if you have a function like `foo a = bar b a`, you can rewrite it as `foo = bar b`, because of currying.

Anyhoo, let's implement another function with a left fold before moving on to right folds. I'm sure you all know that `elem` checks whether a value is part of a list so I won't go into that again (whoops, just did!). Let's implement it with a left fold.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Well, well, well, what do we have here? The starting value and accumulator here is a boolean value. The type of the accumulator value and the end result is always the same when dealing with folds. Remember that if you ever don't know what to use as a starting value, it'll give you some idea. We start off with `False`. It makes sense to use `False` as a starting value. We assume it isn't there. Also, if we call a fold on an empty list, the result will just be the starting value. Then we check the current element is the element we're looking for. If it is, we set the accumulator to `True`. If it's not, we just leave the accumulator unchanged. If it was `False` before, it stays that way because this current element is not it. If it was `True`, we leave it at that.

The right fold, `foldr` works in a similar way to the left fold, only the accumulator eats up the values from the right. Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so `\acc x -> ...`), the right fold's binary function has the current value as the first parameter and the accumulator as the second one (so `\x acc -> ...`). It kind of makes sense that the right fold has the accumulator on the right, because it folds from the right side.

The accumulator value (and hence, the result) of a fold can be of any type. It can be a number, a boolean or even a new list. We'll be implementing the map function with a right fold. The accumulator will be a list, we'll be accumulating the mapped list element by element. From that, it's obvious that the starting element will be an empty list.

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

If we're mapping `(+3)` to `[1,2,3]`, we approach the list from the right side. We take the last element, which is `3` and apply the function to it, which ends up being `6`. Then, we prepend it to the accumulator, which is was `[]`. `6:[]` is `[6]` and that's now the accumulator. We apply `(+3)` to `2`, that's `5` and we prepend (`:`) it to the accumulator, so the accumulator is now `[5,6]`. We apply `(+3)` to `1` and prepend that to the accumulator and so the end value is `[4,5,6]`.

Of course, we could have implemented this function with a left fold too. It would be `map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs`, but the thing is that the `++` function is much more expensive than `:`, so we usually use right folds when we're building up new lists from a list.

If you reverse a list, you can do a right fold on just like you would have done a left fold and vice versa. Sometimes you don't even have to do that.

The sum function can be implemented pretty much the same with a left and right fold. One big difference is that right folds work on infinite lists, whereas left ones don't! To put it plainly, if you take an infinite list at some point and you fold it up from the right, you'll eventually reach the beginning of the list. However, if you take an infinite list at a point and you try to fold it up from the left, you'll never reach an end!

Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold. That's why folds are, along with maps and filters, one of the most useful types of functions in functional programming.

The `foldl1` and `foldr1` functions work much like `foldl` and `foldr`, only you don't need to provide them with an explicit starting value. They assume the first (or last) element of the list to be the starting value and then start the fold with the element next to it. With that in mind, the sum function can be implemented like so: sum = foldl1 (+). Because they depend on the lists they fold up having at least one element, they cause runtime errors if called with empty lists. foldl and foldr, on the other hand, work fine with empty lists. When making a fold, think about how it acts on an empty list. If the function doesn't make sense when given an empty list, you can probably use a foldl1 or foldr1 to implement it.

Just to show you how powerful folds are, we're going to implement a bunch of standard library functions by using folds:

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
product' :: (Num a) => [a] -> a
product' = foldr1 (*)
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
head' :: [a] -> a
head' = foldr1 (\x _ -> x)
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

head and last are better implemented by just pattern matching, but this just goes to show, you can still achieve them by using folds. Our reverse' definition is pretty clever, I think. We take a starting value of an empty list and then approach our list from the left and just prepend to our accumulator. In the end, we build up a reversed list. \acc x -> x : acc kind of looks like the : function, only the parameters are flipped. That's why we could have also written our reverse as foldl (flip (:)) [].

Another way to picture right and left folds is like this: say we have a right fold and the binary function is f and the starting value is z. If we're right folding over the list [3,4,5,6], we're essentially doing this: f 3 (f 4 (f 5 (f 6 z))). f is called with the last element in the list and the accumulator, that value is given as the accumulator to the next to last value and so on. If we take f to be +

and the starting accumulator value to be `0`, that's `3 + (4 + (5 + (6 + 0)))`. Or if we write `+` as a prefix function, that's `(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))`. Similarly, doing a left fold over that list with `g` as the binary function and `z` as the accumulator is the equivalent of `g (g (g (g z 3) 4) 5) 6`. If we use `flip (:)` as the binary function and `[]` as the accumulator (so we're reversing the list), then that's the equivalent of `flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6`. And sure enough, if you evaluate that expression, you get `[6,5,4,3]`.

`scanl` and `scanr` are like `foldl` and `foldr`, only they report all the intermediate accumulator states in the form of a list. There are also `scanl1` and `scanr1`, which are analogous to `foldl1` and `foldr1`.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

When using a `scanl`, the final result will be in the last element of the resulting list while a `scanr` will place the result in the head.

Scans are used to monitor the progression of a function that can be implemented as a fold. Let's answer us this question: **How many elements does it take for the sum of the roots of all natural numbers to exceed 1000?**. To get the squares of all natural numbers, we just do `map sqrt [1..]`. Now, to get the sum, we could do a fold, but because we're interested in how the sum progresses, we're going to do a scan. Once we've done the scan, we just see how many sums are under 1000. The first sum in the scanlist will be 1, normally. The second will be 1 plus the square root of 2. The third will be that plus the square root of 3. If there are X sums under 1000, then it takes X+1 elements for the sum to exceed 1000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

We use `takeWhile` here instead of `filter` because `filter` doesn't work on infinite lists. Even though we know the list is ascending, `filter` doesn't, so we use `takeWhile` to cut the scanlist off at the first occurence of a sum greater than 1000.

6.6 Function application with $

Alright, next up, we'll take a look at the `$` function, also called *function application*. First of all, let's check out how it's defined:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

What the heck? What is this useless operator? It's just function application! Well, almost, but not quite! Whereas normal function application (putting a space between two things) has a really high precedence, the `$` function has the lowest precedence. Function application with a space is left-associative (so `f a b c` is the same as `((f a) b) c)`), function application with `$` is right-associative.

That's all very well, but how does this help us? Most of the time, it's a convenience function so that we don't have to write so many parentheses. Consider the expression `sum (map sqrt [1..130])`. Because `$` has such a low precedence, we can rewrite that expression as `sum $ map sqrt [1..130]`, saving ourselves precious keystrokes! When a `$` is encountered, the expression on its right is applied as the parameter to the function on its left. How about `sqrt 3 + 4 + 9`? This adds together 9, 4 and the square root of 3. If we want get the square root of *3 + 4 + 9*, we'd have to write `sqrt (3 + 4 + 9)` or if we use `$` we can write it as `sqrt $ 3 + 4 + 9` because `$` has the lowest precedence of any operator. That's why you can imagine a `$` being sort of the equivalent of writing an opening parentheses and then writing a closing one on the far right side of the expression.

How about `sum (filter (> 10) (map (*2) [2..10]))`? Well, because `$` is right-associative, `f (g (z x))` is equal to `f $ g $ z x`. And so, we can rewrite `sum (filter (> 10) (map (*2) [2..10]))` as `sum $ filter (> 10) $ map (*2) [2..10]`.

But apart from getting rid of parentheses, `$` means that function application can be treated just like another function. That way, we can, for instance, map function application over a list of functions.

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

6.7   Function composition

In mathematics, function composition is defined like this: $(f \circ g)(x) = f(g(x))$, meaning that composing two functions produces a new function that, when called with a parameter, say, *x* is the equivalent of calling *g* with the parameter *x* and then calling the *f* with that result.

In Haskell, function composition is pretty much the same thing. We do function composition with the `.` function, which is defined like so:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Mind the type declaration. `f` must take as its parameter a value that has the same type as `g`'s return value. So the resulting function takes a parameter of the same type that `g` takes and returns a value of the same type that `f` returns. The expression `negate . (* 3)` returns a function that takes a number,

multiplies it by 3 and then negates it.

One of the uses for function composition is making functions on the fly to pass to other functions. Sure, can use lambdas for that, but many times, function composition is clearer and more concise. Say we have a list of numbers and we want to turn them all into negative numbers. One way to do that would be to get each number's absolute value and then negate it, like so:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Notice the lambda and how it looks like the result function composition. Using function composition, we can rewrite that as:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Fabulous! Function composition is right-associative, so we can compose many functions at a time. The expression `f (g (z x))` is equivalent to `(f . g . z) x`. With that in mind, we can turn

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

into

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

But what about functions that take several parameters? Well, if we want to use them in function composition, we usually have to partially apply them just so much that each function takes just one parameter. `sum (replicate 5 (max 6.7 8.9))` can be rewritten as `(sum . replicate 5 . max 6.7) 8.9` or as `sum . replicate 5 . max 6.7 $ 8.9`. What goes on in here is this: a function that takes what `max 6.7` takes and applies `replicate 5` to it is created. Then, a function that takes the result of that and does a sum of it is created. Finally, that function is called with `8.9`. But normally, you just read that as: apply `8.9` to `max 6.7`, then apply `replicate 5` to that and then apply `sum` to that. If you want to rewrite an expression with a lot of parentheses by using function composition, you can start by putting the last parameter of the innermost function after a `$` and then just composing all the other function calls, writing them without their last parameter and putting dots between them. If you have `replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))`, you can write it as `replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]`. If the expression ends with three parentheses, chances are that if you translate it into function composition, it'll have three composition operators.

Another common use of function composition is defining functions in the so-called point free style (also called the point*less* style). Take for example this function that we wrote earlier:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

The `xs` is exposed on both right sides. Because of currying, we can omit the `xs` on both sides, because calling `foldl (+) 0` creates a function that takes a list. Writing the function as `sum' = foldl (+) 0` is called writing it in point free style. How would we write this in point free style?

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

We can't just get rid of the `x` on both right right sides. The `x` in the function body has parentheses after it. `cos (max 50)` wouldn't make sense. You can't get the cosine of a function. What we can do is express `fn` as a composition of functions.

```
fn = ceiling . negate . tan . cos . max 50
```

Excellent! Many times, a point free style is more readable and concise, because it makes you think about functions and what kind of functions composing them results in instead of thinking about data and how it's shuffled around. You can take simple functions and use composition as glue to form more complex functions. However, many times, writing a function in point free style can be less readable if a function is too complex. That's why making long chains of function composition is discouraged, although I plead guilty of sometimes being too composition-happy. The prefered style is to use *let* bindings to give labels to intermediary results or split the problem into sub-problems and then put it together so that the function makes sense to someone reading it instead of just making a huge composition chain.

In the section about maps and filters, we solved a problem of finding the sum of all odd squares that are smaller than 10,000. Here's what the solution looks like when put into a function.

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Being such a fan of function composition, I would have probably written that like this:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

However, if there was a chance of someone else reading that code, I would have written it like this:

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

It wouldn't win any code golf competition, but someone reading the function will probably find it easier to read than a composition chain.

Chapter 7

Modules

7.1 Loading modules

A Haskell module is a collection of related functions, types and typeclasses. A Haskell program is a collection of modules where the main module loads up the other modules and then uses the functions defined in them to do something. Having code split up into several modules has quite a lot of advantages. If a module is generic enough, the functions it exports can be used in a multitude of different programs. If your own code is separated into self-contained modules which don't rely on each other too much (we also say they are loosely coupled), you can reuse them later on. It makes the whole deal of writing code more manageable by having it split into several parts, each of which has some sort of purpose.

The Haskell standard library is split into modules, each of them contains functions and types that are somehow related and serve some common purpose. There's a module for manipulating lists, a module for concurrent programming, a module for dealing with complex numbers, etc. All the functions, types and typeclasses that we've dealt with so far were part of the `Prelude` module, which is imported by default. In this chapter, we're going to examine a few useful modules and the functions that they have. But first, we're going to see how to import modules.

The syntax for importing modules in a Haskell script is `import <module name>`. This must be done before defining any functions, so imports are usually done at the top of the file. One script can, of course, import several modules. Just put each import statement into a separate line. Let's import the `Data.List` module, which has a bunch of useful functions for working with lists and use a function that it exports to create a function that tells us how many unique elements a list has.

```
import Data.List
numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

When you do `import Data.List`, all the functions that `Data.List` exports become available in the global namespace, meaning that you can call them

from wherever in the script. `nub` is a function defined in `Data.List` that takes a list and weeds out duplicate elements. Composing `length` and `nub` by doing `length . nub` produces a function that's the equivalent of `\xs -> length (nub xs)`.

You can also put the functions of modules into the global namespace when using GHCI. If you're in GHCI and you want to be able to call the functions exported by `Data.List`, do this:

```
ghci> :m + Data.List
```

If we want to load up the names from several modules inside GHCI, we don't have to do `:m +` several times, we can just load up several modules at once.

```
ghci> :m + Data.List Data.Map Data.Set
```

However, if you've loaded a script that already imports a module, you don't need to use `:m +` to get access to it.

If you just need a couple of functions from a module, you can selectively import just those functions. If we wanted to import only the `nub` and `sort` functions from `Data.List`, we'd do this:

```
import Data.List (nub, sort)
```

You can also choose to import all of the functions of a module except a few select ones. That's often useful when several modules export functions with the same name and you want to get rid of the offending ones. Say we already have our own function that's called `nub` and we want to import all the functions from `Data.List` except the `nub` function:

```
import Data.List hiding (nub)
```

Another way of dealing with name clashes is to do qualified imports. The `Data.Map` module, which offers a data structure for looking up values by key, exports a bunch of functions with the same name as `Prelude` functions, like `filter` or `null`. So when we import `Data.Map` and then call `filter`, Haskell won't know which function to use. Here's how we solve this:

```
import qualified Data.Map
```

This makes it so that if we want to reference `Data.Map`'s `filter` function, we have to do `Data.Map.filter`, whereas just `filter` still refers to the normal `filter` we all know and love. But typing out `Data.Map` in front of every function from that module is kind of tedious. That's why we can rename the qualified import to something shorter:

```
import qualified Data.Map as M
```

Now, to reference `Data.Map`'s `filter` function, we just use `M.filter`.

Use this handy reference to see which modules are in the standard library. A great way to pick up new Haskell knowledge is to just click through the standard library reference and explore the modules and their functions. You can also view the Haskell source code for each module. Reading the source code of some modules is a really good way to learn Haskell and get a solid feel for it.

To search for functions or to find out where they're located, use Hoogle. It's a really awesome Haskell search engine, you can search by name, module name or even type signature.

7.2 Data.List

The `Data.List` module is all about lists, obviously. It provides some very useful functions for dealing with them. We've already met some of its functions (like `map` and `filter`) because the `Prelude` module exports some functions from `Data.List` for convenience. You don't have to import `Data.List` via a qualified import because it doesn't clash with any `Prelude` names except for those that `Prelude` already steals from `Data.List`. Let's take a look at some of the functions that we haven't met before.

`intersperse` takes an element and a list and then puts that element in between each pair of elements in the list. Here's a demonstration:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

`intercalate` takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

`transpose` transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Say we have the polynomials $3x^2 + 5x + 9$, $10x^3 + 9$ and $8x^3 + 5x^2 + x - 1$ and we want to add them together. We can use the lists `[0,3,5,9]`, `[10,0,0,9]` and `[8,5,1,-1]` to represent them in Haskell. Now, to add them, all we have to do is this:

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

When we transpose these three lists, the third powers are then in the first row, the second powers in the second one and so on. Mapping `sum` to that produces our desired result.

`foldl'`, `foldl1'`, `foldr'` and `foldr1'` are strict versions of their respective lazy incarnations. When using lazy folds on really big lists, you might often get a stack overflow error. The culprit for that is that due to the lazy nature of the folds, the accumulator value isn't actually updated as the folding happens. What actually happens is that the accumulator kind of makes a promise that it will compute its value when asked to actually produce the result. That

happens for every intermediate accumulator and all those promises overflow
your stack. The strict folds aren't lazy buggers and actually compute the
intermediate values as they go along instead of filling up your stack with
promises. So if you ever get stack overflow errors when doing lazy folds, try
switching to their strict versions.

`concat` flattens a list of lists into just a list of elements.

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

It will just remove one level of nesting. So if you want to completely flatten
[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]], which is a list of lists of lists, you have to
concatenate it twice.

Doing `concatMap` is the same as first mapping a function to a list and then
concatenating the list with `concat`.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

`and` takes a list of boolean values and returns `True` only if all the values in
the list are `True`.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

`or` is like `and`, only it returns `True` if any of the boolean values in a list is
`True`.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` and `all` take a predicate and then check if any or all the elements in
a list satisfy the predicate, respectively. Usually we use these two functions
instead of mapping over a list and then doing `and` or `or`.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all ('elem' ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any ('elem' ['A'..'Z']) "HEYGUYSwhatsup"
True
```

`iterate` takes a function and a starting value. It applies the function to
the starting value, then it applies that function to the result, then it applies
the function to that result again, etc. It returns all the results in the form of
an infinite list.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

`splitAt` takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

`takeWhile` is a really useful little function. It takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it's cut off. It turns out this is very useful.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Say we wanted to know the sum of all third powers that are under 10,000. We can't map `(^3)` to `[1..]`, apply a filter and then try to sum that up because filtering an infinite list never finishes. You may know that all the elements here are ascending but Haskell doesn't. That's why we can do this:

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

We apply `(^3)` to an infinite list and then once an element that's over 10,000 is encountered, the list is cut off. Now we can sum it up easily.

`dropWhile` is similar, only it drops all the elements while the predicate is true. Once predicate equates to `False`, it returns the rest of the list. An extremely useful and lovely function!

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

We're given a list that represents the value of a stock by date. The list is made of tuples whose first component is the stock value, the second is the year, the third is the month and the fourth is the date. We want to know when the stock value first exceeded one thousand dollars!

```
ghci> let stock =
[(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(998.3,
2008,9,5)]
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
(1001.4,2008,9,4)
```

`span` is kind of like `takeWhile`, only it returns a pair of lists. The first list contains everything the resulting list from `takeWhile` would contain if it were called with the same predicate and the same list. The second list contains the part of the list that would have been dropped.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in "First word:" ++
fw ++ ", the rest:" ++ rest
"First word: This, the rest: is a sentence"
```

Whereas `span` spans the list while the predicate is true, `break` breaks it when the predicate is first true. Doing `break p` is the equivalent of doing `span (not . p)`.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

When using `break`, the second list in the result will start with the first element that satisfies the predicate.

`sort` simply sorts a list. The type of the elements in the list has to be part of the `Ord` typeclass, because if the elements of a list can't be put in some kind of order, then the list can't be sorted.

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

`group` takes a list and groups adjacent elements into sublists if they are equal.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

If we sort a list before grouping it, we can find out how many times each element appears in the list.

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $
[1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

`inits` and `tails` are like `init` and `tail`, only they recursively apply that to a list until there's nothing left. Observe.

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Let's use a fold to implement searching a list for a sublist.

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -> if take nlen x == needle then True else acc) False
(tails haystack)
```

First we call `tails` with the list in which we're searching. Then we go over each tail and see if it starts with what we're looking for.

With that, we actually just made a function that behaves like `isInfixOf`. `isInfixOf` searches for a sublist within a list and returns `True` if the sublist we're looking for is somewhere inside the target list.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
```

```
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

isPrefixOf and isSuffixOf search for a sublist at the beginning and at the
end of a list, respectively.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

elem and notElem check if an element is or isn't inside a list.

partition takes a list and a predicate and returns a pair of lists. The first
list in the result contains all the elements that satisfy the predicate, the second
contains all the ones that don't.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

It's important to understand how this is different from span and break:

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

While span and break are done once they encounter the first element that
doesn't and does satisfy the predicate, partition goes through the whole list
and splits it up according to the predicate.

find takes a list and a predicate and returns the first element that satisfies
the predicate. But it returns that element wrapped in a Maybe value. We'll
be covering algebraic data types more in depth in the next chapter but for
now, this is what you need to know: a Maybe value can either be Just something
or Nothing. Much like a list can be either an empty list or a list with some
elements, a Maybe value can be either no elements or a single element. And
like the type of a list of, say, integers is [Int], the type of maybe having an
integer is Maybe Int. Anyway, let's take our find function for a spin.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Notice the type of find. Its result is Maybe a. That's kind of like having the
type of [a], only a value of the type Maybe can contain either no elements or
one element, whereas a list can contain no elements, one element or several
elements.

Remember when we were searching for the first time our stock went over
$1000. We did head (dropWhile (\(val,y,m,d) -> val < 1000) stock). Remember

that `head` is not really safe. What would happen if our stock never went
over $1000? Our application of `dropWhile` would return an empty list and
getting the head of an empty list would result in an error. However, if we
rewrote that as `find (\(val,y,m,d) -> val > 1000) stock`, we'd be much safer. If
our stock never went over $1000 (so if no element satisfied the predicate),
we'd get back a `Nothing`. But there was a valid answer in that list, we'd get,
say, `Just (1001.4,2008,9,4)`.

`elemIndex` is kind of like `elem`, only it doesn't return a boolean value. It
maybe returns the index of the element we're looking for. If that element isn't
in our list, it returns a `Nothing`.

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` is like `elemIndex`, only it returns a list of indices, in case the
element we're looking for crops up in our list several times. Because we're
using a list to represent the indices, we don't need a `Maybe` type, because failure
can be represented as the empty list, which is very much synonymous to
`Nothing`.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

`findIndex` is like find, but it maybe returns the index of the first element
that satisfies the predicate. `findIndices` returns the indices of all elements
that satisfy the predicate in the form of a list.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

We already covered `zip` and `zipWith`. We noted that they zip together two
lists, either in a tuple or with a binary function (meaning such a function that
takes two parameters). But what if we want to zip together three lists? Or
zip three lists with a function that takes three parameters? Well, for that, we
have `zip3`, `zip4`, etc. and `zipWith3`, `zipWith4`, etc. These variants go up to 7.
While this may look like a hack, it works out pretty fine, because there aren't
many times when you want to zip 8 lists together. There's also a very clever
way for zipping infinite numbers of lists, but we're not advanced enough to
cover that just yet.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Just like with normal zipping, lists that are longer than the shortest list
that's being zipped are cut down to size.

`lines` is a useful function when dealing with files or input from somewhere. It takes a string and returns every line of that string in a separate list.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

`'n'` is the character for a unix newline. Backslashes have special meaning in Haskell strings and characters.

`unlines` is the inverse function of `lines`. It takes a list of strings and joins them together using a `'n'`.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` and `unwords` are for splitting a line of text into words or joining a list of words into a text. Very useful.

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these          are     the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

We've already mentioned `nub`. It takes a list and weeds out the duplicate elements, returning a list whose every element is a unique snowflake! The function does have a kind of strange name. It turns out that "nub" means a small lump or essential part of something. In my opinion, they should use real words for function names instead of old-people words.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

`delete` takes an element and a list and deletes the first occurence of that element in the list.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

`\\` is the list difference function. It acts like a set difference, basically. For every element in the right-hand list, it removes a matching element in the left one.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

Doing `[1..10] \ [2,5,9]` is like doing `delete 2 . delete 5 . delete 9 $ [1..10]`.

`union` also acts like a function on sets. It returns the union of two lists. It pretty much goes over every element in the second list and appends it to the first one if it isn't already in yet. Watch out though, duplicates are removed from the second list!

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

intersect works like set intersection. It returns only the elements that are found in both lists.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

insert takes an element and a list of elements that can be sorted and inserts it into the list. It will insert it so that all the elements to its right are greater than or equal to it. If we use insert to insert into a sorted list, the resulting list will be kept sorted.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

What length, take, drop, splitAt, !! and replicate have in common is that they take an Int as one of their parameters, even though they could be more generic and usable if they just took any type that's part of the Integral or Num typeclasses (depending on the functions). They do that for historical reasons. However, fixing that would probably break a lot of existing code. That's why Data.List has their more generic equivalents, named genericLength, genericTake, genericDrop, genericSplitAt, genericIndex and genericReplicate. For instance, length has a type signature of length :: [a] -> Int. If we try to get the average of a list of numbers by doing let xs = [1..6] in sum xs / length xs, we get a type error, because you can't use / with an Int. genericLength, on the other hand, has a type signature of genericLength :: (Num a) =>[b] -> a. Because a Num can act like a floating point number, getting the average by doing let xs = [1..6] in sum xs / genericLength xs works out just fine.

The nub, delete, union, intersect and group functions all have their more general counterparts called nubBy, deleteBy, unionBy, intersectBy and groupBy. The difference between them is that the first set of functions use == to test for equality, whereas the *By* ones also take an equality function and then compare them by using that equality function. group is the same as groupBy (==).

For instance, say we have a list that describes the value of a function for every second. We want to segment it into sublists based on when the value was below zero and when it went above. If we just did a normal group, it would just group the equal adjacent values together. But what we want is to group them by whether they are negative or not. That's where groupBy comes in! The equality function supplied to the *By* functions should take two elements of the same type and return True if it considers them equal by its standards.

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4,
-14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

From this, we clearly see which sections are positive and which are negative. The equality function supplied takes two elements and then returns `True` only if they're both negative or if they're both positive. This equality function can also be written as `xy -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`, although I think the first way is more readable. An even clearer way to write equality functions for the *By* functions is if you import the `on` function from `Data.Function`. `on` is defined like this:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f 'on' g = \x y -> f (g x) (g y)
```

So doing `(==) 'on' (> 0)` returns an equality function that looks like `xy -> (x > 0) == (y > 0)`. `on` is used a lot with the *By* functions because with it, we can do:

```
ghci> groupBy ((==) 'on' (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Very readable indeed! You can read it out loud: Group this by equality on whether the elements are greater than zero.

Similarly, the `sort`, `insert`, `maximum` and `minimum` also have their more general equivalents. Functions like `groupBy` take a function that determines when two elements are equal. `sortBy`, `insertBy`, `maximumBy` and `minimumBy` take a function that determine if one element is greater, smaller or equal to the other. The type signature of `sortBy` is `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. If you remember from before, the `Ordering` type can have a value of `LT`, `EQ` or `GT`. `sort` is the equivalent of `sortBy compare`, because compare just takes two elements whose type is in the `Ord` typeclass and returns their ordering relationship.

Lists can be compared, but when they are, they are compared lexicographically. What if we have a list of lists and we want to sort it not based on the inner lists' contents but on their lengths? Well, as you've probably guessed, we'll use the `sortBy` function.

```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare 'on' length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

Awesome! `compare 'on' length` ... man, that reads almost like real English! If you're not sure how exactly the `on` works here, `compare 'on' length` is the equivalent of `xy -> length x 'compare' length y`. When you're dealing with *By* functions that take an equality function, you usually do `(==) 'on' something` and when you're dealing with *By* functions that take an ordering function, you usually do `compare 'on' something`.

7.3 Data.Char

The `Data.Char` module does what its name suggests. It exports functions that deal with characters. It's also helpful when filtering and mapping over strings because they're just lists of characters.

`Data.Char` exports a bunch of predicates over characters. That is, functions that take a character and tell us whether some assumption about it is true or

false. Here's what they are:

`isControl` checks whether a character is a control character.

`isSpace` checks whether a character is a white-space characters. That includes spaces, tab characters, newlines, etc.

`isLower` checks whether a character is lower-cased.

`isUpper` checks whether a character is upper-cased.

`isAlpha` checks whether a character is a letter.

`isAlphaNum` checks whether a character is a letter or a number.

`isPrint` checks whether a character is printable. Control characters, for instance, are not printable.

`isDigit` checks whether a character is a digit.

`isOctDigit` checks whether a character is an octal digit.

`isHexDigit` checks whether a character is a hex dibit.

`isLetter` checks whether a character is a letter.

`isMark` checks for Unicode mark characters. Those are characters that combine with preceding letters to form latters with accents. Use this if you are French.

`isNumber` checks whether a character is numeric.

`isPunctuation` checks whether a character is punctuation.

`isSymbol` checks whether a character is a fancy mathematical or currency symbol.

`isSeparator` checks for Unicode spaces and separators.

`isAscii` checks whether a character falls into the first 128 characters of the Unicode character set.

`isLatin1` checks whether a character falls into the first 256 characters of Unicode.

`isAsciiUpper` checks whether a character is ASCII and upper-case.

`isAsciiLower` checks whether a character is ASCII and lower-case.

All these predicates have a type signature of `Char -> Bool`. Most of the time you'll use this to filter out strings or something like that. For instance, let's say we're making a program that takes a username and the username can only be comprised of alphanumeric characters. We can use the `Data.List` function `all` in combination with the `Data.Char` predicates to determine if the username is alright.

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Kewl. In case you don't remember, `all` takes a predicate and a list and returns `True` only if that predicate holds for every element in the list.

We can also use `isSpace` to simulate the `Data.List` function `words`.

```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
ghci>
```

Hmmm, well, it kind of does what `words` does but we're left with elements of only spaces. Hmm, whatever shall we do? I know, let's filter that sucker.

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hey guys
its me"
["hey","guys","its","me"]
```

Ah.

The `Data.Char` also exports a datatype that's kind of like `Ordering`. The `Ordering` type can have a value of `LT`, `EQ` or `GT`. It's a sort of enumeration. It describes a few possible results that can arise from comparing two elements. The `GeneralCategory` type is also an enumeration. It presents us with a few possible categories that a character can fall into. The main function for getting the general category of a character is `generalCategory`. It has a type of `generalCategory :: Char -> GeneralCategory`. There are about 31 categories so we won't list them all here, but let's play around with the function.

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSym
bol]
```

Since the `GeneralCategory` type is part of the `Eq` typeclass, we can also test for stuff like `generalCategory c == Space`.

`toUpper` converts a character to upper-case. Spaces, numbers, and the like remain unchanged.

`toLower` converts a character to lower-case.

`toTitle` converts a character to title-case. For most characters, title-case is the same as upper-case.

`digitToInt` converts a character to an `Int`. To succeed, the character must be in the ranges `'0'..'9'`, `'a'..'f'` or `'A'..'F'`.

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` is the inverse function of `digitToInt`. It takes an `Int` in the range of `0..15` and converts it to a lower-case character.

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

The `ord` and `chr` functions convert characters to their corresponding numbers and vice versa:

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

The difference between the `ord` values of two characters is equal to how far apart they are in the Unicode table.

The Caesar cipher is a primitive method of encoding messages by shifting each character in them by a fixed number of positions in the alphabet. We can easily create a sort of Caesar cipher of our own, only we won't constrict ourselves to the alphabet.

```
encode :: Int -> String -> String
encode shift msg =
    let ords = map ord msg
        shifted = map (+ shift) ords
    in  map chr shifted
```

Here, we first convert the string to a list of numbers. Then we add the shift amount to each number before converting the list of numbers back to characters. If you're a composition cowboy, you could write the body of this function as `map (chr . (+ shift) . ord) msg`. Let's try encoding a few messages.

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

That's encoded alright. Decoding a message is basically just shifting it back by the number of places it was shifted by in the first place.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

7.4 Data.Map

Association lists (also called dictionaries) are lists that are used to store key-value pairs where ordering doesn't matter. For instance, we might use an association list to store phone numbers, where phone numbers would be the values and people's names would be the keys. We don't care in which order they're stored, we just want to get the right phone number for the right person.

The most obvious way to represent association lists in Haskell would be by having a list of pairs. The first component in the pair would be the key, the second component the value. Here's an example of an association list with phone numbers:

```
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

Despite this seemingly odd indentation, this is just a list of pairs of strings. The most common task when dealing with association lists is looking up some value by key. Let's make a function that looks up some value given a key.

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```

Pretty simple. The function that takes a key and a list, filters the list so that only matching keys remain, gets the first key-value that matches and returns the value. But what happens if the key we're looking for isn't in the association list? Hmm. Here, if a key isn't in the association list, we'll end up trying to get the head of an empty list, which throws a runtime error. However, we should avoid making our programs so easy to crash, so let's use the Maybe data type. If we don't find the key, we'll return a Nothing. If we find it, we'll return Just something, where something is the value corresponding to that key.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                            then Just v
                            else findKey key xs
```

Look at the type declaration. It takes a key that can be equated, an association list and then it maybe produces a value. Sounds about right.

This is a textbook recursive function that operates on a list. Edge case, splitting a list into a head and a tail, recursive calls, they're all there. This is the classic fold pattern, so let's see how this would be implemented as a fold.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

> **Note:** It's usually better to use folds for this standard list recursion pattern instead of explicitly writing the recursion because they're easier to read and identify. Everyone knows it's a fold when they see the `foldr` call, but it takes some more thinking to read explicit recursion.

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```

Works like a charm! If we have the girl's phone number, we `Just` get the number, otherwise we get `Nothing`.

We just implemented the `lookup` function from `Data.List`. If we want to find the corresponding value to a key, we have to traverse all the elements of the list until we find it. The `Data.Map` module offers association lists that are much faster (because they're internally implemented with trees) and also it provides a lot of utility functions. From now on, we'll say we're working with maps instead of association lists.

Because `Data.Map` exports functions that clash with the `Prelude` and `Data.List` ones, we'll do a qualified import.

```
import qualified Data.Map as Map
```

Put this import statement into a script and then load the script via GHCI.

Let's go ahead and see what `Data.Map` has in store for us! Here's the basic rundown of its functions.

The `fromList` function takes an association list (in the form of a list) and returns a map with the same associations.

```
ghci> Map.fromList
[("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

If there are duplicate keys in the original association list, the duplicates are just discarded. This is the type signature of `fromList`

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

It says that it takes a list of pairs of type `k` and `v` and returns a map that maps from keys of type `k` to type `v`. Notice that when we were doing association lists with normal lists, the keys only had to be equatable (their type belonging to the `Eq` typeclass) but now they have to be orderable. That's an essential constraint in the `Data.Map` module. It needs the keys to be orderable so it can arrange them in a tree.

You should always use `Data.Map` for key-value associations unless you have keys that aren't part of the `Ord` typeclass.

`empty` represents an empty map. It takes no arguments, it just returns an empty map.

```
ghci> Map.empty
fromList []
```

`insert` takes a key, a value and a map and returns a new map that's just like the old one, only with the key and value inserted.

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

We can implement our own `fromList` by using the empty map, `insert` and a fold. Watch:

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

It's a pretty straightforward fold. We start of with an empty map and we fold it up from the right, inserting the key value pairs into the accumulator as we go along.

`null` checks if a map is empty.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

`size` reports the size of a map.

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

`singleton` takes a key and a value and creates a map that has exactly one mapping.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

`lookup` works like the `Data.List` `lookup`, only it operates on maps. It returns `Just something` if it finds something for the key and `Nothing` if it doesn't.

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

`member` is a predicate takes a key and a map and reports whether the key is in the map or not.

`map` and `filter` work much like their list equivalents.

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

`toList` is the inverse of `fromList`.

```
ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

`keys` and `elems` return lists of keys and values respectively. `keys` is the equivalent of `map fst . Map.toList` and `elems` is the equivalent of `map snd . Map.toList`.

`fromListWith` is a cool little function. It acts like `fromList`, only it doesn't discard duplicate keys but it uses a function supplied to it to decide what to do with them. Let's say that a girl can have several numbers and we have an association list set up like this.

```
phoneBook =
    [("betty","555-2938")
    ,("betty","342-2492")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("patsy","943-2929")
    ,("patsy","827-9162")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ,("penny","555-2111")
    ]
```

Now if we just use `fromList` to put that into a map, we'll lose a few numbers! So here's what we'll do:

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++
number2) xs
```

`insertWith` is to `insert` what `fromListWith` is to `fromList`. It inserts a key-value pair into a map, but if that map already contains the key, it uses the function passed to it to determine what to do.

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

These were just a few functions from `Data.Map`. You can see a complete list of functions in the documentation.

7.5 Data.Set

The `Data.Set` module offers us, well, sets. Like sets from mathematics. Sets are kind of like a cross between lists and maps. All the elements in a set are unique. And because they're internally implemented with trees (much like maps in `Data.Map`), they're ordered. Checking for membership, inserting, deleting, etc. is much faster than doing the same thing with lists. The most common operation when dealing with sets are inserting into a set, checking for membership and converting a set to a list.

Because the names in `Data.Set` clash with a lot of `Prelude` and `Data.List` names, we do a qualified import.

Put this import statement in a script:

```
import qualified Data.Set as Set
```

And then load the script via GHCI.

Let's say we have two pieces of text. We want to find out which characters were used in both of them.

```
text1 = "I just had an anime dream. Anime... Reality... Are they so
different?"
text2 = "The old man left his garbage can out and now his trash is all over
my lawn!"
```

The `fromList` function works much like you would expect. It takes a list and converts it into a set.

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
fromList " !Tabcdefghilmnorstuvwy"
```

As you can see, the items are ordered and each element is unique. Now let's use the `intersection` function to see which elements they both share.

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

We can use the `difference` function to see which letters are in the first set but aren't in the second one and vice versa.

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

Or we can see all the unique letters used in both sentences by using `union`.

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

The `null`, `size`, `member`, `empty`, `singleton`, `insert` and `delete` functions all work like you'd expect them to.

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

We can also check for subsets or proper subset. Set A is a subset of set B if B contains all the elements that A does. Set A is a proper subset of set B if B contains all the elements that A does but has more elements.

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList
[1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

We can also `map` over sets and `filter` them.

```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

Sets are often used to weed a list of duplicates from a list by first making it into a set with `fromList` and then converting it back to a list with `toList`. The `Data.List` function `nub` already does that, but weeding out duplicates for large lists is much faster if you cram them into a set and then convert them back to a list than using `nub`. But using `nub` only requires the type of the list's elements to be part of the `Eq` typeclass, whereas if you want to cram elements into a set, the type of the list has to be in `Ord`.

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

`setNub` is generally faster than `nub` on big lists but as you can see, `nub` preserves the ordering of the list's elements, while `setNub` does not.

7.6   Making our own modules

We've looked at some cool modules so far, but how do we make our own module? Almost every programming language enables you to split your code up into several files and Haskell is no different. When making programs, it's good practice to take functions and types that work towards a similar purpose

and put them in a module. That way, you can easily reuse those functions in other programs by just importing your module.

Let's see how we can make our own modules by making a little module that provides some functions for calculating the volume and area of a few geometrical objects. We'll start by creating a file called `Geometry.hs`.

We say that a module *exports* functions. What that means is that when I import a module, I can use the functions that it exports. It can define functions that its functions call internally, but we can only see and use the ones that it exports.

At the beginning of a module, we specify the module name. If we have a file called `Geometry.hs`, then we should name our module `Geometry`. Then, we specify the functions that it exports and after that, we can start writing the functions. So we'll start with this.

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

As you can see, we'll be doing areas and volumes for spheres, cubes and cuboids. Let's go ahead and define our functions then:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)
sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)
cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side
cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side
cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c
cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 +
rectangleArea c b * 2
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Pretty standard geometry right here. There are a few things to take note of though. Because a cube is only a special case of a cuboid, we defined its area and volume by treating it as a cuboid whose sides are all of the same length. We also defined a helper function called `rectangleArea`, which calculates a rectangle's area based on the lenghts of its sides. It's rather trivial because

it's just multiplication. Notice that we used it in our functions in the module
(namely `cuboidArea` and `cuboidVolume`) but we didn't export it! Because we want
our module to just present functions for dealing with three dimensional objects,
we used `rectangleArea` but we didn't export it.

When making a module, we usually export only those functions that act
as a sort of interface to our module so that the implementation is hidden. If
someone is using our `Geometry` module, they don't have to concern themselves
with functions that we don't export. We can decide to change those functions
completely or delete them in a newer version (we could delete `rectangleArea`
and just use `*` instead) and no one will mind because we weren't exporting
them in the first place.

To use our module, we just do:

```
import Geometry
```

`Geometry.hs` has to be in the same folder that the program that's importing
it is in, though.

Modules can also be given a hierarchical structures. Each module can
have a number of sub-modules and they can have sub-modules of their own.
Let's section these functions off so that `Geometry` is a module that has three
sub-modules, one for each type of object.

First, we'll make a folder called `Geometry`. Mind the capital G. In it, we'll
place three files: `sphere.hs`, `cuboid.hs`, and `cube.hs`. Here's what the files will
contain:

sphere.hs

```
module Geometry.Sphere
( volume
, area
) where
volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)
area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

cuboid.hs

```
module Geometry.Cuboid
( volume
, area
) where
volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c
area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c
b * 2
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

cube.hs

```
module Geometry.Cube
( volume
```

```
, area
) where
import qualified Geometry.Cuboid as Cuboid
volume :: Float -> Float
volume side = Cuboid.volume side side side
area :: Float -> Float
area side = Cuboid.area side side side
```

Alright! So first is `Geometry.Sphere`. Notice how we placed it in a folder called `Geometry` and then defined the module name as `Geometry.Sphere`. We did the same for the cuboid. Also notice how in all three sub-modules, we defined functions with the same names. We can do this because they're separate modules. We want to use functions from `Geometry.Cuboid` in `Geometry.Cube` but we can't just straight up do `import Geometry.Cuboid` because it exports functions with the same names as `Geometry.Cube`. That's why we do a qualified import and all is well.

So now if we're in a file that's on the same level as the `Geometry` folder, we can do, say:

```
import Geometry.Sphere
```

And then we can call `area` and `volume` and they'll give us the area and volume for a sphere. And if we want to juggle two or more of these modules, we have to do qualified imports because they export functions with the same names. So we just do something like:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

And then we can call `Sphere.area`, `Sphere.volume`, `Cuboid.area`, etc. and each will calculate the area or volume for their corresponding object.

The next time you find yourself writing a file that's really big and has a lot of functions, try to see which functions serve some common purpose and then see if you can put them in their own module. You'll be able to just import your module the next time you're writing a program that requires some of the same functionality.

Chapter 8

Making Our Own Types and Typeclasses

In the previous chapters, we covered some existing Haskell types and typeclasses. In this chapter, we'll learn how to make our own and how to put them to work!

8.1   Algebraic data types intro

So far, we've run into a lot of data types. `Bool`, `Int`, `Char`, `Maybe`, etc. But how do we make our own? Well, one way is to use the **data** keyword to define a type. Let's see how the `Bool` type is defined in the standard library.

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

`data` means that we're defining a new data type. The part before the `=` denotes the type, which is `Bool`. The parts after the `=` are **value constructors**. They specify the different values that this type can have. The `|` is read as *or*. So we can read this as: the `Bool` type can have a value of `True` or `False`. Both the type name and the value constructors have to be capital cased.

In a similar fashion, we can think of the `Int` type as being defined like this:

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... |
2147483647
```

The first and last value constructors are the minimum and maximum possible values of `Int`. It's not actually defined like this, the ellipses are here because we omitted a heapload of numbers, so this is just for illustrative purposes.

Now, let's think about how we would represent a shape in Haskell. One way would be to use tuples. A circle could be denoted as `(43.1, 55.0, 10.4)` where the first and second fields are the coordinates of the circle's center and the third field is the radius. Sounds OK, but those could also represent a 3D vector or anything else. A better solution would be to make our own type to represent a shape. Let's say that a shape can be a circle or a rectangle. Here it is:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Now what's this? Think of it like this. The `Circle` value constructor has three fields, which take floats. So when we write a value constructor, we can optionally add some types after it and those types define the values it will contain. Here, the first two fields are the coordinates of its center, the third one its radius. The `Rectangle` value constructor has four fields which accept floats. The first two are the coordinates to its upper left corner and the second two are coordinates to its lower right one.

Now when I say fields, I actually mean parameters. Value constructors are actually functions that ultimately return a value of a data type. Let's take a look at the type signatures for these two value constructors.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Cool, so value constructors are functions like everything else. Who would have thought? Let's make a function that takes a shape and returns its surface.

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

The first notable thing here is the type declaration. It says that the function takes a shape and returns a float. We couldn't write a type declaration of `Circle -> Float` because `Circle` is not a type, `Shape` is. Just like we can't write a function with a type declaration of `True -> Int`. The next thing we notice here is that we can pattern match against constructors. We pattern matched against constructors before (all the time actually) when we pattern matched against values like `[]` or `False` or `5`, only those values didn't have any fields. We just write a constructor and then bind its fields to names. Because we're interested in the radius, we don't actually care about the first two fields, which tell us where the circle is.

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Yay, it works! But if we try to just print out `Circle 10 20 5` in the prompt, we'll get an error. That's because Haskell doesn't know how to display our data type as a string (yet). Remember, when we try to print a value out in the prompt, Haskell first runs the `show` function to get the string representation of our value and then it prints that out to the terminal. To make our `Shape` type part of the `Show` typeclass, we modify it like this:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
deriving (Show)
```

We won't concern ourselves with deriving too much for now. Let's just say that if we add `deriving (Show)` at the end of a *data* declaration, Haskell automagically makes that type part of the `Show` typeclass. So now, we can do this:

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Value constructors are functions, so we can map them and partially apply them and everything. If we want a list of concentric circles with different radii, we can do this.

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0
20.0 6.0]
```

Our data type is good, although it could be better. Let's make an intermediate data type that defines a point in two-dimensional space. Then we can use that to make our shapes more understandable.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

Notice that when defining a point, we used the same name for the data type and the value constructor. This has no special meaning, although it's common to use the same name as the type if there's only one value constructor. So now the `Circle` has two fields, one is of type `Point` and the other of type `Float`. This makes it easier to understand what's what. Same goes for the rectangle. We have to adjust our `surface` function to reflect these changes.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2
- y1)
```

The only thing we had to change were the patterns. We disregarded the whole point in the circle pattern. In the rectangle pattern, we just used a nested pattern matching to get the fields of the points. If we wanted to reference the points themselves for some reason, we could have used as-patterns.

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

How about a function that nudges a shape? It takes a shape, the amount to move it on the x axis and the amount to move it on the y axis and then returns a new shape that has the same dimensions, only it's located somewhere else.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a)
(y1+b)) (Point (x2+a) (y2+b))
```

Pretty straightforward. We add the nudge amounts to the points that denote the position of the shape.

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

If we don't want to deal directly with points, we can make some auxilliary functions that create shapes of some size at the zero coordinates and then nudge those.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r
baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

You can, of course, export your data types in your modules. To do that, just write your type along with the functions you are exporting and then add some parentheses and in them specify the value constructors that you want to export for it, separated by commas. If you want to export all the value constructors for a given type, just write ... If we wanted to export the functions and types that we defined here in a module, we could start it off like this:

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

By doing `Shape(..)`, we exported all the value constructors for `Shape`, so that means that whoever imports our module can make shapes by using the `Rectangle` and `Circle` value constructors. It's the same as writing `Shape (Rectangle, Circle)`.

We could also opt not to export any value constructors for `Shape` by just writing `Shape` in the export statement. That way, someone importing our module could only make shapes by using the auxilliary functions `baseCircle` and `baseRect`. `Data.Map` uses that approach. You can't create a map by doing `Map.Map [(1,2),(3,4)]` because it doesn't export that value constructor. However, you can make a mapping by using one of the auxilliary functions like `Map.fromList`. Remember, value constructors are just functions that take the fields as parameters and return a value of some type (like `Shape`) as a result. So when we choose not to export them, we just prevent the person importing our module from using those functions, but if some other functions that are exported return a type, we can use them to make values of our custom data types.

Not exporting the value constructors of a data types makes them more abstract in such a way that we hide their implementation. Also, whoever uses our module can't pattern match against the value constructors.

8.2   Record syntax

OK, we've been tasked with creating a data type that describes a person. The info that we want to store about that person is: first name, last name, age, height, phone number, and favorite ice-cream flavor. I don't know about you, but that's all I ever want to know about a person. Let's give it a go!

```
data Person = Person String String Int Float String String deriving (Show)
```

O-kay. The first field is the first name, the second is the last name, the third is the age and so on. Let's make a person.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

That's kind of cool, although slightly unreadable. What if we want to create a function to get seperate info from a person? A function that gets some person's first name, a function that gets some person's last name, etc. Well, we'd have to define them kind of like this.

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname
lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname
age :: Person -> Int
age (Person _ _ age _ _ _) = age
height :: Person -> Float
height (Person _ _ _ height _ _) = height
phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number
flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Whew! I certainly did not enjoy writing that! Despite being very cumbersome and BORING to write, this method works.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

There must be a better way, you say! Well no, there isn't, sorry.

Just kidding, there is. Hahaha! The makers of Haskell were very smart and anticipated this scenario. They included an alternative way to write data types. Here's how we could achieve the above functionality with record syntax.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: 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, `firstName` and then we write a double colon `::` (also called Paamayim Nekudotayim, haha) 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: `firstName`, `lastName`, `age`, `height`, `phoneNumber` and `flavor`.

```
ghci> :t flavor
flavor :: Person -> String
```

```
ghci> :t firstName
firstName :: 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. Say we have a type that represents a car. We want to keep track of the company that made it, the model name and its year of production. Watch.

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

If we define it using record syntax, we can make a new car like this.

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

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

When making a new car, we don't have to necessarily put the fields in the proper order, as long as we list all of them. But if we don't use record syntax, we have to specify them in order. Use record syntax when a constructor has several fields and it's not obvious which field is which. If we make a 3D vector data type by doing `data Vector = Vector Int Int Int`, it's pretty obvious that the fields are the components of a vector. However, in our `Person` and `Car` types, it wasn't so obvious and we greatly benefited from using record syntax.

8.3 Type parameters

A value 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. If you're familiar with templates in C++, you'll see some parallels. 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. That's why we can do `[1,2,3] ++ []` and `["ha","ha","ha"] ++ []`.

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 that we've already met 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 comparing 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 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. 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
```

8.4   Derived instances

In the Typeclasses 101 section, we explained the basics of typeclasses. We explained that a typeclass is a sort of an interface that defines some behavior. A type can be made an **instance** of a typeclass if it supports that behavior.

Example: the `Int` type is an instance of the `Eq` typeclass because the `Eq` typeclass defines behavior for stuff that can be equated. And because integers can be equated, `Int` is a part of the `Eq` typeclass. The real usefulness comes with the functions that act as the interface for `Eq`, namely `==` and `/=`. If a type is a part of the `Eq` typeclass, we can use the `==` functions with values of that type. That's why expressions like `4 == 4` and `"foo" /= "bar"` typecheck.

We also mentioned that they're often confused with classes in languages like Java, Python, C++ and the like, which then baffles a lot of people. In those languages, classes are a blueprint from which we then create objects that contain state and can do some actions. Typeclasses are more like interfaces. We don't make data from typeclasses. Instead, we first make our data type and then we think about what it can act like. If it can act like something that can be equated, we make it an instance of the `Eq` typeclass. If it can act like something that can be ordered, we make it an instance of the `Ord` typeclass.

In the next section, we'll take a look at how we can manually make our types instances of typeclasses by implementing the functions defined by the typeclasses. But right now, let's see how Haskell can automatically make our type an instance of any of the following typeclasses: `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`. Haskell can derive the behavior of our types in these contexts if we use the *deriving* keyword when making our data type.

Consider this data type:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

It describes a person. Let's assume that no two people have the same combination of first name, last name and age. Now, if we have records for two people, does it make sense to see if they represent the same person? Sure it does. We can try to equate them and see if they're equal or not. That's why it would make sense for this type to be part of the `Eq` typeclass. We'll derive the instance.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

When we derive the `Eq` instance for a type and then try to compare two values of that type with `==` or `/=`, Haskell will see if the value constructors match (there's only one value constructor here though) and then it will check if all the data contained inside matches by testing each pair of fields with `==`. There's only one catch though, the types of all the fields also have to be part of the `Eq` typeclass. But since both `String` and `Int` are, we're OK. Let's test our `Eq` instance.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age =
43}
ghci> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age =
41}
ghci> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
```

```
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

Of course, since `Person` is now in `Eq`, we can use it as the `a` for all functions that have a class constraint of `Eq a` in their type signature, such as `elem`.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

The `Show` and `Read` typeclasses are for things that can be converted to or from strings, respectively. Like with `Eq`, if a type's constructors have fields, their type has to be a part of `Show` or `Read` if we want to make our type an instance of them. Let's make our `Person` data type a part of `Show` and `Read` as well.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Now we can print a person out to the terminal.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age =
43}
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age =
43}"
```

Had we tried to print a person on the terminal before making the `Person` data type part of `Show`, Haskell would have complained at us, claiming it doesn't know how to represent a person as a string. But now that we've derived a `Show` instance for it, it does know.

`Read` is pretty much the inverse typeclass of `Show`. `Show` is for converting values of our a type to a string, `Read` is for converting strings to values of our type. Remember though, when we use the `read` function, we have to use an explicit type annotation to tell Haskell which type we want to get as a result. If we don't make the type we want as a result explicit, Haskell doesn't know which type we want.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
:: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

If we use the result of our `read` later on in a way that Haskell can infer that it should read it as a person, we don't have to use type annotation.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
== mikeD
True
```

We can also read parameterized types, but we have to fill in the type parameters. So we can't do `read "Just 't'":: Maybe a`, but we can do `read "Just 't'":: Maybe Char`.

Derived instances of `Ord` work like expected. First the constructors are compared lexicographically and if the values of two constructors are the same, their fields are compared, provided that the types of the fields are also instances of `Ord`. The `Bool` type can have a value of either `False` or `True`. For the purpose of seeing how it behaves when compared, we can think of it as being implemented like this:

```
data Bool = False | True deriving (Ord)
```

Because the `False` value constructor is specified first and the `True` value constructor is specified after it, we can consider `True` as greater than `False`.

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
False
```

In the `Maybe a` data type, the `Nothing` value constructor is specified before the `Just` value constructor, so a value of `Nothing` is always smaller than a value of `Just something`, even if that something is minus one billion trillion. But if we compare two `Just` values, then it goes to compare what's inside them.

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

But we can't do something like `Just (*3) > Just (*2)`, because `(*3)` and `(*2)` are functions, which aren't instances of `Ord`.

We can easily use algebraic data types to make enumerations and the `Enum` and `Bounded` typeclasses help us with that. Consider the following data type:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
Sunday
```

Because all the value constructors are nullary (take no parameters, i.e. fields), we can make it part of the `Enum` typeclass. The `Enum` typeclass is for things that have predecessors and successors. We can also make it part of the `Bounded` typeclass, which is for things that have a lowest possible value and highest possible value. And while we're at it, let's also make it an instance of all the other derivable typeclasses and see what we can do with it.

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
Sunday
           deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Because it's part of the `Show` and `Read` typeclasses, we can convert values of this type to and from strings.

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Because it's part of the `Eq` and `Ord` typeclasses, we can compare or equate days.

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

It's also part of `Bounded`, so we can get the lowest and highest day.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

It's also an instance of `Enum`. We can get predecessors and successors of days and we can make list ranges from them!

```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

That's pretty awesome.

8.5 Type synonyms

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.

When we were dealing with the `Data.Map` module, we first represented a phonebook with an association list before converting it into a map. As we've already found out, an association list is a list of key-value pairs. Let's look at a phonebook that we had.

```
phoneBook :: [(String,String)]
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

We see that the type of `phoneBook` is `[(String,String)]`. That tells us that it's an association list that maps from strings to strings, but not much else. Let's make a type synonym to convey some more information in the type declaration.

```
type PhoneBook = [(String,String)]
```

Now the type declaration for our phonebook can be `phoneBook :: PhoneBook`. Let's make a type synonym for `String` as well.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Giving the `String` type synonyms is something that Haskell programmers do when they want to convey more information about what strings in their functions should be used as and what they represent.

So now, when we implement a function that takes a name and a number and sees if that name and number combination is in our phonebook, we can give it a very pretty and descriptive type declaration.

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

If we decided not to use type synonyms, our function would have a type of `String -> String -> [(String,String)] -> Bool`. In this case, the type declaration that took advantage of type synonyms is easier to understand. However, you shouldn't go overboard with them. We introduce type synonyms either to describe what some existing type represents in our functions (and thus our type declarations become better documentation) or when something has a long-ish type that's repeated a lot (like `[(String,String)]`) but represents something more specific in the context of our functions.

Type synonyms can also be parameterized. If we want a type that represents an association list type but still want it to be general so it can use any type as the keys and values, we can do this:

```
type AssocList k v = [(k,v)]
```

Now, a function that gets the value by a key in an association list can have a type of `(Eq k) =>k -> AssocList k v -> Maybe v`. `AssocList` is a type constructor that takes two types and produces a concrete type, like `AssocList Int String`, for instance.

> **Fonzie says:** Aaay! When I talk about *concrete types* I mean like fully applied types like `Map Int String` or if we're dealin' with one of them polymorphic functions, `[a]` or `(Ord a) =>Maybe a` and stuff. And like, sometimes me and the boys say that `Maybe` is a type, but we don't mean that, cause every idiot knows `Maybe` is a type constructor. When I apply an extra type to `Maybe`, like `Maybe String`, then I have a concrete type. You know, values can only have types that are concrete types! So in conclusion, live fast, love hard and don't let anybody else use your comb!

Just like we can partially apply functions to get new functions, we can partially apply type parameters and get new type constructors from them. Just like we call a function with too few parameters to get back a new function, we can specify a type constructor with too few type parameters and get back a partially applied type constructor. If we wanted a type that represents a map (from `Data.Map`) from integers to something, we could either do this:

```
type IntMap v = Map Int v
```

Or we could do it like this:

```
type IntMap = Map Int
```

Either way, the `IntMap` type constructor takes one parameter and that is the type of what the integers will point to.

> **Oh yeah**. If you're going to try and implement this, you'll probably going to do a qualified import of `Data.Map`. When you do a qualified import, type constructors also have to be preceeded with a module name. So you'd write `type IntMap = Map.Map Int`.

Make sure that you really understand the distinction between type constructors and value constructors. Just because we made a type synonym called `IntMap` or `AssocList` doesn't mean that we can do stuff like `AssocList [(1,2),(4,5),(7,9)]`. All it means is that we can refer to its type by using different names. We can do `[(1,2),(3,5),(8,9)] :: AssocList Int Int`, which will make the numbers inside assume a type of `Int`, but we can still use that list as we would any normal list that has pairs of integers inside. Type synonyms (and types generally) can only be used in the type portion of Haskell. We're in Haskell's type portion whenever we're defining new types (so in *data* and *type* declarations) or when we're located after a `::`. The `::` is in type declarations or in type annotations.

Another cool data type that takes two types as its parameters is the `Either a b` type. This is roughly how it's defined:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

It has two value constructors. If the `Left` is used, then its contents are of type `a` and if `Right` is used, then its contents are of type `b`. So we can use this type to encapsulate a value of one type or another and then when we get a value of type `Either a b`, we usually pattern match on both `Left` and `Right` and we different stuff based on which one of them it was.

```
ghci> Right 20
Right 20
```

```
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

So far, we've seen that `Maybe a` was mostly used to represent the results of computations that could have either failed or not. But somtimes, `Maybe a` isn't good enough because `Nothing` doesn't really convey much information other than that something has failed. That's cool for functions that can fail in only one way or if we're just not interested in how and why they failed. A `Data.Map` lookup fails only if the key we were looking for wasn't in the map, so we know exactly what happened. However, when we're interested in how some function failed or why, we usually use the result type of `Either a b`, where `a` is some sort of type that can tell us something about the possible failure and `b` is the type of a successful computation. Hence, errors use the `Left` value constructor while results use `Right`.

An example: a high-school has lockers so that students have some place to put their Guns'n'Roses posters. Each locker has a code combination. When a student wants a new locker, they tell the locker supervisor which locker number they want and he gives them the code. However, if someone is already using that locker, he can't tell them the code for the locker and they have to pick a different one. We'll use a map from `Data.Map` to represent the lockers. It'll map from locker numbers to a pair of whether the locker is in use or not and the locker code.

```
import qualified Data.Map as Map
data LockerState = Taken | Free deriving (Show, Eq)
type Code = String
type LockerMap = Map.Map Int (LockerState, Code)
```

Simple stuff. We introduce a new data type to represent whether a locker is taken or free and we make a type synonym for the locker code. We also make a type synonym for the type that maps from integers to pairs of locker state and code. And now, we're going to make a function that searches for the code in a locker map. We're going to use an `Either String Code` type to represent our result, because our lookup can fail in two ways — the locker can be taken, in which case we can't tell the code or the locker number might not exist at all. If the lookup fails, we're just going to use a `String` to tell what's happened.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't
exist!"
        Just (state, code) -> if state /= Taken
                                then Right code
                                else Left $ "Locker " ++ show lockerNumber ++
" is already taken!"
```

We do a normal lookup in the map. If we get a `Nothing`, we return a value of type `Left String`, saying that the locker doesn't exist at all. If we do find it, then we do an additional check to see if the locker is taken. If it is, return

a `Left` saying that it's already taken. If it isn't, then return a value of type `Right Code`, in which we give the student the correct code for the locker. It's actually a `Right String`, but we introduced that type synonym to introduce some additional documentation into the type declaration. Here's an example map:

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

Now let's try looking up some locker codes.

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

We could have used a `Maybe a` to represent the result but then we wouldn't know why we couldn't get the code. But now, we have information about the failure in our result type.

8.6   Recursive data structures

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.

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]`.

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)
```

This reads just like our definition of lists from one of the previous paragraphs. It's either an empty list or a combination of a head with some value and a list. If you're confused about this, you might find it easier to understand in record syntax.

```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving
(Show, Read, Eq, Ord)
```

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:[])`.

We can define functions to be automatically infix by making them comprised of only special characters. We can also do the same with constructors, since they're just functions that return a data type. So check this out.

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

First off, we notice a new syntactic construct, the fixity declarations. When we define functions as operators, we can use that to give them a fixity (but we don't have to). A fixity states how tightly the operator binds and whether it's left-associative or right-associative. For instance, `*`'s fixity is `infixl 7 *` and `+`'s fixity is `infixl 6`. That means that they're both left-associative (`4 * 3 * 2` is `(4 * 3) * 2`) but `*` binds tighter than `+`, because it has a greater fixity, so `5 * 4 + 3` is `(5 * 4) + 3`.

Otherwise, we just wrote `a :-: (List a)` instead of `Cons a (List a)`. Now, we can write out lists in our list type like so:

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

When deriving `Show` for our type, Haskell will still display it as if the constructor was a prefix function, hence the parentheses around the operator (remember, `4 + 3` is `(+) 4 3`).

Let's make a function that adds two of our lists together. This is how `++` is defined for normal lists:

```
infixr 5  ++
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

So we'll just steal that for our own list. We'll name the function `.++`.

```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

And let's see if it works ...

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

Nice. Is nice. If we wanted, we could implement all of the functions that operate on lists on our own list type.

Notice how we pattern matched on `(x :-: xs)`. That works because pattern matching is actually about matching constructors. We can match on `:-:` because it is a constructor for our own list type and we can also match on `:` because it is a constructor for the built-in list type. Same goes for `[]`. Because pattern matching works (only) on constructors, we can match for stuff like that, normal prefix constructors or stuff like `8` or `'a'`, which are basically constructors for the numeric and character types, respectively.

Now, we're going to implement a **binary search tree**. 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 sub-tree of, say, 5 are going to be smaller than 5. Elements in its right sub-tree 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.

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)
```

Okay, good, this is good. Instead of manually building a tree, we're going to make a function that takes a tree and an element and inserts an element.

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.

In languages like C, we'd do this by modifying the pointers and values inside the tree. 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 doesn't really have a concept of pointer, just values. 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 are two functions. One is a utility function for making a singleton tree (a tree with just one node) and a function to insert an element into a tree.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
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)
```

The `singleton` function is just a shortcut for making a node that has something and then two empty sub-trees. In the insertion function, we first have the edge condition as a pattern. If we've reached an empty sub-tree, that means we're where we want and instead of the empty tree, we put a singleton tree 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 sub-tree but instead of its left sub-tree, 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.

Next up, we're going to make a function that checks if some element is in the tree. First, let's define the edge condition. If we're looking for an element in an empty tree, then it's certainly not there. Okay. Notice how this is the same as the edge condition when searching for elements in lists. If we're looking for an element in an empty list, it's not there. Anyway, if we're not looking for an element in an 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 sub-tree. If it's bigger, check to see if it's in the right sub-tree.

```
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
```

All we had to do was write up the previous paragraph in code. Let's have some fun with our trees! Instead of manually building one (although we could), we'll use a fold to build up a tree from a list. Remember, pretty much everything that traverses a list one by one and then returns some sort of value can be implemented with a fold! We're going to start with the empty tree and then approach a list from the right and just insert element after element into our accumulator tree.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree))
(Node 7 (Node 6 EmptyTree EmptyTree) (Node 8 EmptyTree EmptyTree))
```

In that `foldr`, `treeInsert` was the folding function (it takes a tree and a list element and produces a new tree) and `EmptyTree` was the starting accumulator. `nums`, of course, was the list we were folding over.

When we print our tree to the console, it's not very readable, but if we try, we can make out its structure. We see that the root node is 5 and then it has two sub-trees, one of which has the root node of 3 and the other a 7, etc.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

Checking for membership also works nicely. Cool.

So as you can see, algebraic data structures are a really cool and powerful concept in Haskell. We can use them to make anything from boolean values and weekday enumerations to binary search trees and more!

8.7   Typeclasses 102

So far, we've learned about some of the standard Haskell typeclasses and we've seen which types are in them. We've also learned how to automatically make our own types instances of the standard typeclasses by asking Haskell to derive the instances for us. In this section, we're going to learn how to make our own typeclasses and how to make types instances of them by hand.

A quick recap on typeclasses: typeclasses are like interfaces. A typeclass defines some behavior (like comparing for equality, comparing for ordering, enumeration) and then types that can behave in that way are made instances of that typeclass. The behavior of typeclasses is achieved by defining functions or just type declarations that we then implement. So when we say that a type is an instance of a typeclass, we mean that we can use the functions that the typeclass defines with that type.

Typeclasses have pretty much nothing to do with classes in languages like Java or Python. This confuses many people, so I want you to forget everything you know about classes in imperative languages right now.

For example, the `Eq` typeclass is for stuff that can be equated. It defines the functions `==` and `/=`. If we have a type (say, `Car`) and comparing two cars with the equality function `==` makes sense, then it makes sense for `Car` to be an instance of `Eq`.

This is how the `Eq` class is defined in the standard prelude:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Woah, woah, woah! Some new strange syntax and keywords there! Don't worry, this will all be clear in a second. First off, when we write `class Eq a where`, this means that we're defining a new typeclass and that's called `Eq`. The `a` is the type variable and it means that `a` will play the role of the type that we will soon be making an instance of `Eq`. It doesn't have to be called `a`, it doesn't even have to be one letter, it just has to be a lowercase word. Then, we define several functions. It's not mandatory to implement the function bodies themselves, we just have to specify the type declarations for the functions.

> Some people might understand this better if we wrote `class Eq equatable where` and then specified the type declarations like `(==) :: equatable -> equatable -> Bool`.

Anyway, we *did* implement the function bodies for the functions that `Eq` defines, only we defined them in terms of mutual recursion. We said that two instances of `Eq` are equal if they are not different and they are different if they are not equal. We didn't have to do this, really, but we did and we'll see how this helps us soon.

> If we have say `class Eq a where` and then define a type declaration within that class like `(==) :: a -> -a -> Bool`, then when we examine the type of that function later on, it will have the type of `(Eq a) =>a -> a -> Bool`.

So once we have a class, what can we do with it? Well, not much, really. But once we start making types instances of that class, we start getting some nice functionality. So check out this type:

```
data TrafficLight = Red | Yellow | Green
```

It defines the states of a traffic light. Notice how we didn't derive any class instances for it. That's because we're going to write up some instances by hand, even though we could derive them for types like `Eq` and `Show`. Here's how we make it an instance of `Eq`.

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

We did it by using the *instance* keyword. So *class* is for defining new typeclasses and *instance* is for making our types instances of typeclasses. When we were defining `Eq`, we wrote `class Eq a where` and we said that `a` plays the role of whichever type will be made an instance later on. We can see that clearly here, because when we're making an instance, we write `instance Eq TrafficLight where`. We replace the `a` with the actual type.

Because `==` was defined in terms of `/=` and vice versa in the *class* declaration, we only had to overwrite one of them in the instance declaration. That's called the minimal complete definition for the typeclass — the minimum of functions that we have to implement so that our type can behave like the class advertises. To fulfill the minimal complete definition for `Eq`, we have to overwrite either one of `==` or `/=`. If `Eq` was defined simply like this:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

We'd have to implement both of these functions when making a type an instance of it, because Haskell wouldn't know how these two functions are related. The minimal complete definition would then be: both `==` and `/=`.

You can see that we implemented `==` simply by doing pattern matching. Since there are many more cases where two lights aren't equal, we specified the ones that are equal and then just did a catch-all pattern saying that if it's none of the previous combinations, then two lights aren't equal.

Let's make this an instance of `Show` by hand, too. To satisfy the minimal complete definition for `Show`, we just have to implement its `show` function, which takes a value and turns it into a string.

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Once again, we used pattern matching to achieve our goals. Let's see how it works in action:

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Nice. We could have just derived `Eq` and it would have had the same effect (but we didn't for educational purposes). However, deriving `Show` would have just directly translated the value constructors to strings. But if we want lights to appear like `"Red light"`, then we have to make the instance declaration by hand.

You can also make typeclasses that are subclasses of other typeclasses. The *class* declaration for `Num` is a bit long, but here's the first part:

```
class (Eq a) => Num a where
   ...
```

As we mentioned previously, there are a lot of places where we can cram in class constraints. So this is just like writing `class Num a where`, only we state that our type `a` must be an instance of `Eq`. We're essentially saying that we have to make a type an instance of `Eq` before we can make it an instance of `Num`. Before some type can be considered a number, it makes sense that we can determine whether values of that type can be equated or not. That's all there is to subclassing really, it's just a class constraint on a *class* declaration! When defining function bodies in the *class* declaration or when defining them in *instance* declarations, we can assume that `a` is a part of `Eq` and so we can use `==` on values of that type.

But how are the `Maybe` or list types made as instances of typeclasses? What makes `Maybe` different from, say, `TrafficLight` is that `Maybe` in itself isn't a concrete type, it's a type constructor that takes one type parameter (like `Char` or something) to produce a concrete type (like `Maybe Char`). Let's take a look at the `Eq` typeclass again:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

From the type declarations, we see that the `a` is used as a concrete type because all the types in functions have to be concrete (remember, you can't have a function of the type `a -> Maybe` but you can have a function of `a -> Maybe a` or `Maybe Int -> Maybe String`). That's why we can't do something like

```
instance Eq Maybe where
   ...
```

Because like we've seen, the `a` has to be a concrete type but `Maybe` isn't a concrete type. It's a type constructor that takes one parameter and then produces a concrete type. It would also be tedious to write `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where`, etc. for every type ever. So we could write it out like so:

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

This is like saying that we want to make all types of the form `Maybe something` an instance of `Eq`. We actually could have written `(Maybe something)`, but we usually opt for single letters to be true to the Haskell style. The `(Maybe m)` here plays the role of the `a` from `class Eq a where`. While `Maybe` isn't a concrete type, `Maybe m` is. By specifying a type parameter (`m`, which is in lowercase), we said that we want all types that are in the form of `Maybe m`, where `m` is any type, to be an instance of `Eq`.

There's one problem with this though. Can you spot it? We use `==` on the contents of the `Maybe` but we have no assurance that what the `Maybe` contains can be used with `Eq`! That's why we have to modify our *instance* declaration like this:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

We had to add a class constraint! With this *instance* declaration, we say this: we want all types of the form `Maybe m` to be part of the `Eq` typeclass, but only those types where the `m` (so what's contained inside the `Maybe`) is also a part of `Eq`. This is actually how Haskell would derive the instance too.

Most of the times, class constraints in *class* declarations are used for making a typeclass a subclass of another typeclass and class constraints in *instance* declarations are used to express requirements about the contents of some type. For instance, here we required the contents of the `Maybe` to also be part of the `Eq` typeclass.

When making instances, if you see that a type is used as a concrete type in the type declarations (like the `a` in `a -> a -> Bool`), you have to supply type parameters and add parentheses so that you end up with a concrete type.

> Take into account that the type you're trying to make an instance of will replace the parameter in the *class* declaration. The `a` from `class Eq a where` will be replaced with a real type when you make an instance, so try mentally putting your type into the function type declarations as well. `(==) :: Maybe -> Maybe -> Bool` doesn't make much sense but `(==) :: (Eq m) =>Maybe m -> Maybe m -> Bool` does. But this is just something to think about, because `==` will always have a type of `(==) :: (Eq a) =>a -> a -> Bool`, no matter what instances we make.

Ooh, one more thing, check this out! If you want to see what the instances of a typeclass are, just do `:info YourTypeClass` in GHCI. So typing `:info Num` will show which functions the typeclass defines and it will give you a list of the types in the typeclass. `:info` works for types and type constructors too. If you do `:info Maybe`, it will show you all the typeclasses that `Maybe` is an instance of. Also `:info` can show you the type declaration of a function. I think that's pretty cool.

8.8   A yes-no typeclass

In JavaScript and some other weakly typed languages, you can put almost anything inside an if expression. For example, you can do all of the following: `if (0) alert("YEAH!") else alert("NO!")`, `if ("") alert ("YEAH!") else alert("NO!")`, `if (false) alert("YEAH") else alert("NO!")`, etc. and all of these will throw an alert of `NO!`. If you do `if ("WHAT") alert ("YEAH") else alert("NO!")`, it will alert a `"YEAH!"` because JavaScript considers non-empty strings to be a sort of true-ish value.

Even though strictly using `Bool` for boolean semantics works better in Haskell, let's try and implement that JavaScript-ish behavior anyway. For fun! Let's start out with a *class* declaration.

```
class YesNo a where
    yesno :: a -> Bool
```

Pretty simple. The `YesNo` typeclass defines one function. That function takes one value of a type that can be considered to hold some concept of true-ness and tells us for sure if it's true or not. Notice that from the way we use the `a` in the function, `a` has to be a concrete type.

Next up, let's define some instances. For numbers, we'll assume that (like in JavaScript) any number that isn't 0 is true-ish and 0 is false-ish.

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

Empty lists (and by extensions, strings) are a no-ish value, while non-empty lists are a yes-ish value.

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

Notice how we just put in a type parameter `a` in there to make the list a concrete type, even though we don't make any assumptions about the type that's contained in the list. What else, hmm ... I know, `Bool` itself also holds true-ness and false-ness and it's pretty obvious which is which.

```
instance YesNo Bool where
    yesno = id
```

Huh? What's `id`? It's just a standard library function that takes a parameter and returns the same thing, which is what we would be writing here anyway.

Let's make `Maybe a` an instance too.

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```

We didn't need a class constraint because we made no assumptions about the contents of the `Maybe`. We just said that it's true-ish if it's a `Just` value and false-ish if it's a `Nothing`. We still had to write out `(Maybe a)` instead of just `Maybe` because if you think about it, a `Maybe -> Bool` function can't exist (because `Maybe` isn't a concrete type), whereas a `Maybe a -> Bool` is fine and dandy. Still, this is really cool because now, any type of the form `Maybe something` is part of `YesNo` and it doesn't matter what that `something` is.

Previously, we defined a `Tree a` type, that represented a binary search tree. We can say an empty tree is false-ish and anything that's not an empty tree is true-ish.

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

Can a traffic light be a yes or no value? Sure. If it's red, you stop. If it's green, you go. If it's yellow? Eh, I usually run the yellows because I live for adrenaline.

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _   = True
```

Cool, now that we have some instances, let's go play!

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

Right, it works! Let's make a function that mimics the if statement, but it
works with `YesNo` values.

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult = if yesno yesnoVal then yesResult else
noResult
```

Pretty straightforward. It takes a yes-no-ish value and two things. If
the yes-no-ish value is more of a yes, it returns the first of the two things,
otherwise it returns the second of them.

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

8.9   The Functor typeclass

So far, we've encountered a lot of the typeclasses in the standard library. We've
played with `Ord`, which is for stuff that can be ordered. We've palled around
with `Eq`, which is for things that can be equated. We've seen `Show`, which
presents an interface for types whose values can be displayed as strings. Our
good friend `Read` is there whenever we need to convert a string to a value of
some type. And now, we're going to take a look at the ¡span class="label
class"¿Functor¡/span¿ typeclass, which is basically for things that can be
mapped over. You're probably thinking about lists now, since mapping over
lists is such a dominant idiom in Haskell. And you're right, the list type is

part of the `Functor` typeclass.

What better way to get to know the `Functor` typeclass than to see how it's implemented? Let's take a peek.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Alright. We see that it defines one function, `fmap`, and doesn't provide any default implementation for it. The type of `fmap` is interesting. In the definitions of typeclasses so far, the type variable that played the role of the type in the typeclass was a concrete type, like the `a` in `(==) :: (Eq a) =>a -> a -> Bool`. But now, the `f` is not a concrete type (a type that a value can hold, like `Int`, `Bool` or `Maybe String`), but a type constructor that takes one type parameter. A quick refresher example: `Maybe Int` is a concrete type, but `Maybe` is a type constructor that takes one type as the parameter. Anyway, we see that `fmap` takes a function from one type to another and a functor applied with one type and returns a functor applied with another type.

If this sounds a bit confusing, don't worry. All will be revealed soon when we check out a few examples. Hmm, this type declaration for `fmap` reminds me of something. If you don't know what the type signature of `map` is, it's: `map :: (a -> b) -> [a] -> [b]`.

Ah, interesting! It takes a function from one type to another and a list of one type and returns a list of another type. My friends, I think we have ourselves a functor! In fact, `map` is just a `fmap` that works only on lists. Here's how the list is an instance of the `Functor` typeclass.

```
instance Functor [] where
    fmap = map
```

That's it! Notice how we didn't write `instance Functor [a] where`, because from `fmap :: (a -> b) -> f a -> f b`, we see that the `f` has to be a type constructor that takes one type. `[a]` is already a concrete type (of a list with any type inside it), while `[]` is a type constructor that takes one type and can produce types such as `[Int]`, `[String]` or even `[[String]]`.

Since for lists, `fmap` is just `map`, we get the same results when using them on lists.

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

What happens when we `map` or `fmap` over an empty list? Well, of course, we get an empty list. It just turns an empty list of type `[a]` into an empty list of type `[b]`.

Types that can act like a box can be functors. You can think of a list as a box that has an infinite amount of little compartments and they can all be empty, one can be full and the others empty or a number of them can be full. So, what else has the properties of being like a box? For one, the `Maybe a` type.

In a way, it's like a box that can either hold nothing, in which case it has the value of `Nothing`, or it can hold one item, like `"HAHA"`, in which case it has a value of `Just "HAHA"`. Here's how `Maybe` is a functor.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Again, notice how we wrote `instance Functor Maybe where` instead of `instance Functor (Maybe m) where`, like we did when we were dealing with `Maybe` and `YesNo`. `Functor` wants a type constructor that takes one type and not a concrete type. If you mentally replace the `f`s with `Maybe`s, `fmap` acts like a `(a -> b) -> Maybe a -> Maybe b` for this particular type, which looks OK. But if you replace `f` with `(Maybe m)`, then it would seem to act like a `(a -> b) -> Maybe m a -> Maybe m b`, which doesn't make any damn sense because `Maybe` takes just one type parameter.

Anyway, the `fmap` implementation is pretty simple. If it's an empty value of `Nothing`, then just return a `Nothing`. If we map over an empty box, we get an empty box. It makes sense. Just like if we map over an empty list, we get back an empty list. If it's not an empty value, but rather a single value packed up in a `Just`, then we apply the function on the contents of the `Just`.

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Another thing that can be mapped over and made an instance of `Functor` is our `Tree a` type. It can be thought of as a box in a way (holds several or no values) and the `Tree` type constructor takes exactly one type parameter. If you look at `fmap` as if it were a function made only for `Tree`, its type signature would look like `(a -> b) -> Tree a -> Tree b`. We're going to use recursion on this one. Mapping over an empty tree will produce an empty tree. Mapping over a non-empty tree will be a tree consisting of our function applied to the root value and its left and right sub-trees will be the previous sub-trees, only our function will be mapped over them.

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub) = Node (f x) (fmap f leftsub) (fmap f
rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree (Node 20
EmptyTree EmptyTree)))) EmptyTree
```

Nice! Now how about `Either a b`? Can this be made a functor? The `Functor` typeclass wants a type constructor that takes only one type parameter but `Either` takes two. Hmmm! I know, we'll partially apply `Either` by feeding it only one parameter so that it has one free parameter. Here's how `Either a` is a functor in the standard libraries:

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

Well well, what did we do here? You can see how we made `Either a` an instance instead of just `Either`. That's because `Either a` is a type constructor that takes one parameter, whereas `Either` takes two. If `fmap` was specifically for `Either a`, the type signature would then be `(b -> c) -> Either a b -> Either a c` because that's the same as `(b -> c) -> (Either a) b -> (Either a) c`. In the implementation, we mapped in the case of a `Right` value constructor, but we didn't in the case of a `Left`. Why is that? Well, if we look back at how the `Either a b` type is defined, it's kind of like:

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

Well, if we wanted to map one function over both of them, `a` and `b` would have to be the same type. I mean, if we tried to map a function that takes a string and returns a string and the `b` was a string but the `a` was a number, that wouldn't really work out. Also, from seeing what `fmap`'s type would be if it operated only on `Either` values, we see that the first parameter has to remain the same while the second one can change and the first parameter is actualized by the `Left` value constructor.

This also goes nicely with our box analogy if we think of the `Left` part as sort of an empty box with an error message written on the side telling us why it's empty.

Maps from `Data.Map` can also be made a functor because they hold values (or not!). In the case of `Map k v`, `fmap` will map a function `v -> v'` over a map of type `Map k v` and return a map of type `Map k v'`.

> Note, the `'` has no special meaning in types just like it doesn't have special meaning when naming values. It's used to denote things that are similar, only slightly changed.

Try figuring out how `Map k` is made an instance of `Functor` by yourself!

With the `Functor` typeclass, we've seen how typeclasses can represent pretty cool higher-order concepts. We've also had some more practice with partially applying types and making instances. In one of the next chapters, we'll also take a look at some laws that apply for functors.

> **Just one more thing!** Functors should obey some laws so that they may have some properties that we can depend on and not think about too much. If we use `fmap (+1)` over the list `[1,2,3,4]`, we expect the result to be `[2,3,4,5]` and not its reverse, `[5,4,3,2]`. If we use `fmap (a-> a)` (the identity function, which just returns its parameter) over some list, we expect to get back the same list as a result. For example, if we gave the wrong functor instance to our `Tree` type, using `fmap` over a tree where the left sub-tree of a node only has elements that are smaller than the node and the right sub-tree only has nodes that are larger than the node might produce a tree where that's not the case. We'll go over the functor laws in more detail in one of the next chapters.

8.10   Kinds and some type-foo

Type constructors take other types as parameters to eventually produce concrete types. That kind of reminds me of functions, which take values as parameters to produce values. 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. This is all very interesting indeed. In this section, we'll take a look at formally defining how types are applied to type constructors, just like we took a look at formally defining how values are applied to functions by using type declarations. **You don't really have to read this section to continue on your magical Haskell quest** and if you don't understand it, don't worry about it. However, getting this will give you a very thorough understanding of the type system.

So, values like `3`, `"YEAH"` or `takeWhile` (functions are also values, because we can pass them around and such) each have their own type. Types are little labels that values carry so that we can reason about the values. But types have their own little labels, called **kinds**. A kind is more or less the type of a type. This may sound a bit weird and confusing, but it's actually a really cool concept.

What are kinds and what are they good for? Well, let's examine the kind of a type by using the `:k` command in GHCI.

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

A star? How quaint. What does that mean? A `*` means that the type is a concrete type. A concrete type is a type that doesn't take any type parameters and values can only have types that are concrete types. If I had to read `*` out loud (I haven't had to do that so far), I'd say *star* or just *type*.

Okay, now let's see what the kind of `Maybe` is.

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

The `Maybe` type constructor 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 :: *
```

Just like I expected! We applied the type parameter to `Maybe` and got back a concrete type (that's what `* -> *` means. A parallel (although not equivalent, types and kinds are two different things) to this is if we do `:t isUpper` and `:t isUpper 'A'`. `isUpper` has a type of `Char -> Bool` and `isUpper 'A'` has a type of `Bool`, because its value is basically `False`. Both those types, however, have a kind of `*`.

We used `:k` on a type to get its kind, just like we can use `:t` on a value to get its type. Like we said, types are the labels of values and kinds are the

labels of types and there are parallels between the two.

Let's look at another kind.

```
ghci> :k Either
Either :: * -> * -> *
```

Aha, this tells us that `Either` takes two concrete types as type parameters to produce a concrete type. It also looks kind of like a type declaration of a function that takes two values and returns something. Type constructors are curried (just like functions), so we can partially apply them.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

When we wanted to make `Either` a part of the `Functor` typeclass, we had to partially apply it because `Functor` wants types that take only one parameter while `Either` takes two. In other words, `Functor` wants types of kind `* -> *` and so we had to partially apply `Either` to get a type of kind `* -> *` instead of its original kind `* -> * -> *`. If we look at the definition of `Functor` again

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

we see that the `f` type variable is used as a type that takes one concrete type to produce a concrete type. We know it has to produce a concrete type because it's used as the type of a value in a function. And from that, we can deduce that types that want to be friends with `Functor` have to be of kind `* -> *`.

Now, let's do some type-foo. Take a look at this typeclass that I'm just going to make up right now:

```
class Tofu t where
    tofu :: j a -> t a j
```

Man, that looks weird. How would we make a type that could be an instance of that strange typeclass? Well, let's look at what its kind would have to be. Because `j a` is used as the type of a value that the `tofu` function takes as its parameter, `j a` has to have a kind of `*`. We assume `*` for `a` and so we can infer that `j` has to have a kind of `* -> *`. We see that `t` has to produce a concrete value too and that it takes two types. And knowing that `a` has a kind of `*` and `j` has a kind of `* -> *`, we infer that `t` has to have a kind of `* -> (* -> *) -> *`. So it takes a concrete type (`a`), a type constructor that takes one concrete type (`j`) and produces a concrete type. Wow.

OK, so let's make a type with a kind of `* -> (* -> *) -> *`. Here's one way of going about it.

```
data Frank a b  = Frank {frankField :: b a} deriving (Show)
```

How do we know this type has a kind of `* -> (* -> *) - > *`? Well, fields in ADTs are made to hold values, so they must be of kind `*`, obviously. We assume `*` for `a`, which means that `b` takes one type parameter and so its kind is `* -> *`. Now we know the kinds of both `a` and `b` and because they're parameters for `Frank`, we see that `Frank` has a kind of `* -> (* -> *) -> *` The first `*` represents

a and the (* -> *) represents b. Let's make some Frank values and check out
their types.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YES"}
Frank {frankField = "YES"} :: Frank Char []
```

Hmm. Because frankField has a type of form a b, its values must have
types that are of a similar form as well. So they can be Just "HAHA", which
has a type of Maybe [Char] or it can have a value of ['Y','E','S'], which has a
type of [Char] (if we used our own list type for this, it would have a type of
List Char). And we see that the types of the Frank values correspond with the
kind for Frank. [Char] has a kind of * and Maybe has a kind of * -> *. Because
in order to have a value, it has to be a concrete type and thus has to be fully
applied, every value of Frank blah blaah has a kind of *.

Making Frank an instance of Tofu is pretty simple. We see that tofu takes a
j a (so an example type of that form would be Maybe Int) and returns a t a j.
So if we replace Frank with j, the result type would be Frank Int Maybe.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

Not very useful, but we did flex our type muscles. Let's do some more
type-foo. We have this data type:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

And now we want to make it an instance of Functor. Functor wants types
of kind * -> * but Barry doesn't look like it has that kind. What is the kind
of Barry? Well, we see it takes three type parameters, so it's going to be
something -> something -> something -> *. It's safe to say that p is a concrete type
and thus has a kind of *. For k, we assume * and so by extension, t has a
kind of * -> *. Now let's just replace those kinds with the *somethings* that we
used as placeholders and we see it has a kind of (* -> *) -> * -> * -> *. Let's
check that with GHCI.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ah, we were right. How satisfying. Now, to make this type a part of
Functor we have to partially apply the first two type parameters so that we're
left with * -> *. That means that the start of the instance declaration will be:
instance Functor (Barry a b) where. If we look at fmap as if it was made specifi-
cally for Barry, it would have a type of fmap :: (a -> b) -> Barry c d a -> Barry c d b,
because we just replace the Functor's f with Barry c d. The third type parameter
from Barry will have to change and we see that it's conviniently in its own
field.

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

There we go! We just mapped the `f` over the first field.

In this section, we took a good look at how type parameters work and kind of formalized them with kinds, just like we formalized function parameters with type declarations. We saw that there are interesting parallels between functions and type constructors. They are, however, two completely different things. When working on real Haskell, you usually won't have to mess with kinds and do kind inference by hand like we did now. Usually, you just have to partially apply your own type to `* -> *` or `*` when making it an instance of one of the standard typeclasses, but it's good to know how and why that actually works. It's also interesting to see that types have little types of their own. Again, you don't really have to understand everything we did here to read on, but if you understand how kinds work, chances are that you have a very solid grasp of Haskell's type system.

Chapter 9

Input and Output

We've mentioned that Haskell is a purely functional language. Whereas in imperative languages you usually get things done by giving the computer a series of steps to execute, functional programming is more of defining what stuff is. In Haskell, a function can't change some state, like changing the contents of a variable (when a function changes state, we say that the function has *side-effects*). The only thing a function can do in Haskell is give us back some result based on the parameters we gave it. If a function is called two times with the same parameters, it has to return the same result. While this may seem a bit limiting when you're coming from an imperative world, we've seen that it's actually really cool. In an imperative language, you have no guarantee that a simple function that should just crunch some numbers won't burn down your house, kidnap your dog and scratch your car with a potato while crunching those numbers. For instance, when we were making a binary search tree, we didn't insert an element into a tree by modifying some tree in place. Our function for inserting into a binary search tree actually returned a new tree, because it can't change the old one.

While functions being unable to change state is good because it helps us reason about our programs, there's one problem with that. If a function can't change anything in the world, how is it supposed to tell us what it calculated? In order to tell us what it calculated, it has to change the state of an output device (usually the state of the screen), which then emits photons that travel to our brain and change the state of our mind, man.

Do not despair, all is not lost. It turns out that Haskell actually has a really clever system for dealing with functions that have side-effects that neatly separates the part of our program that is pure and the part of our program that is impure, which does all the dirty work like talking to the keyboard and the screen. With those two parts separated, we can still reason about our pure program and take advantage of all the things that purity offers, like laziness, robustness and modularity while efficiently communicating with the outside world.

9.1   Hello, world!

Up until now, we've always loaded our functions into GHCI to test them out and play with them. We've also explored the standard library functions that way. But now, after eight or so chapters, we're finally going to write our first *real* Haskell program! Yay! And sure enough, we're going to do the good old `"hello, world"` schtick.

> **Hey!** For the purposes of this chapter, I'm going to assume you're using a unix-y environment for learning Haskell. If you're in Windows, I'd suggest you download Cygwin, which is a Linux-like environment for Windows, A.K.A. just what you need.

So, for starters, punch in the following in your favorite text editor:

```
main = putStrLn "hello, world"
```

We just defined a name called `main` and in it we call a function called `putStrLn` with the parameter `"hello, world"`. Looks pretty much run of the mill, but it isn't, as we'll see in just a few moments. Save that file as `helloworld.hs`.

And now, we're going to do something we've never done before. We're actually going to compile our program! I'm so excited! Open up your terminal and navigate to the directory where `helloworld.hs` is located and do the following:

```
$ ghc --make helloworld
[1 of 1] Compiling Main               ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

Okay! With any luck, you got something like this and now you can run your program by doing `./helloworld`.

```
$ ./helloworld
hello, world
```

And there we go, our first compiled program that printed out something to the terminal. How extraordinarily boring!

Let's examine what we wrote. First, let's look at the type of the function `putStrLn`.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

We can read the type of `putStrLn` like this: `putStrLn` takes a string and returns an **I/O action** that has a result type of `()` (i.e. the empty tuple, also know as unit). An I/O action is something that, when performed, will carry out an action with a side-effect (that's usually either reading from the input or printing stuff to the screen) and will also contain some kind of return value inside it. Printing a string to the terminal doesn't really have any kind of meaningful return value, so a dummy value of `()` is used.

> The empty tuple is a value of `()` and it also has a type of `()`.

So, when will an I/O action be performed? Well, this is where `main` comes in. An I/O action will be performed when we give it a name of `main` and then run our program.

Having your whole program be just one I/O action seems kind of limiting. That's why we can use *do* syntax to glue together several I/O actions into one. Take a look at the following example:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Ah, interesting, new syntax! And this reads pretty much like an imperative program. If you compile it and try it out, it will probably behave just like you expect it to. Notice that we said *do* and then we laid out a series of steps, like we would in an imperative program. Each of these steps is an I/O action. By putting them together with *do* syntax, we glued them into one I/O action. The action that we got has a type of `IO ()`, because that's the type of the last I/O action inside.

Because of that, `main` always has a type signature of `main :: IO <something>`, where `<something>` is some concrete type. By convention, we don't usually specify a type declaration for `main`.

An interesting thing that we haven't met before is the third line, which states `name <- getLine`. It looks like it reads a line from the input and stores it into a variable called `name`. Does it really? Well, let's examine the type of `getLine`.

```
ghci> :t getLine
getLine :: IO String
```

Aha, o-kay. `getLine` is an I/O action that contains a result type of `String`. That makes sense, because it will wait for the user to input something at the terminal and then that something will be represented as a string. So what's up with `name <- getLine` then? You can read that piece of code like this: **perform the I/O action `getLine` and then bind its result value to** `name`. `getLine` has a type of `IO String`, so `name` will have a type of `String`. You can think of an I/O action as a box with little feet that will go out into the real world and do something there (like write some graffiti on a wall) and maybe bring back some data. Once it's fetched that data for you, the only way to open the box and get the data inside it is to use the `<-` construct. And if we're taking data out of an I/O action, we can only take it out when we're inside another I/O action. This is how Haskell manages to neatly separate the pure and impure parts of our code. `getLine` is in a sense impure because its result value is not guaranteed to be the same when performed twice. That's why it's sort of *tainted* with the `IO` type constructor and we can only get that data out in I/O code. And because I/O code is tainted too, any computation that depends on tainted I/O data will have a tainted result.

When I say *tainted*, I don't mean tainted in such a way that we can never use the result contained in an I/O action ever again in pure code. No, we

temporarily *un-taint* the data inside an I/O action when we bind it to a name.
When we do `name <- getLine`, `name` is just a normal string, because it represents
what's inside the box. We can have a really complicated function that, say,
takes your name (a normal string) as a parameter and tells you your fortune
and your whole life's future based on your name. We can do this:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future: " ++
tellFortune name
```

and `tellFortune` (or any of the functions it passes `name` to) doesn't have to
know anything about I/O, it's just a normal `String -> String` function!

Take a look at this piece of code. Is it valid?

```
nameTag = "Hello, my name is " ++ getLine
```

If you said no, go eat a cookie. If you said yes, drink a bowl of molten
lava. Just kidding, don't! The reason that this doesn't work is that `++` requires
both its parameters to be lists over the same type. The left parameter has a
type of `String` (or `[Char]` if you will), whilst `getLine` has a type of `IO String`. You
can't concatenate a string and an I/O action. We first have to get the result
out of the I/O action to get a value of type `String` and the only way to do that
is to say something like `name <- getLine` inside some other I/O action. If we
want to deal with impure data, we have to do it in an impure environment.
So the taint of impurity spreads around much like the undead scourge and
it's in our best interest to keep the I/O parts of our code as small as possible.

Every I/O action that gets performed has a result encapsulated within it.
That's why our previous example program could also have been written like
this:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

However, `foo` would just have a value of `()`, so doing that would be kind of
moot. Notice that we didn't bind the last `putStrLn` to anything. That's because
in a *do* block, **the last action cannot be bound to a name** like the first two
were. We'll see exactly why that is so a bit later when we venture off into the
world of monads. For now, you can think of it in the way that the *do* block au-
tomatically extracts the value from the last action and binds it to its own result.

Except for the last line, every line in a *do* block that doesn't bind can also be
written with a bind. So `putStrLn "BLAH"` can be written as `_ <- putStrLn "BLAH"`.
But that's useless, so we leave out the `<-` for I/O actions that don't contain an
important result, like `putStrLn <something>`.

Beginners sometimes think that doing

```
name = getLine
```

will read from the input and then bind the value of that to `name`. Well, it
won't, all this does is give the `getLine` I/O action a different name called, well,

`name`. Remember, to get the value out of an I/O action, you have to perform it inside another I/O action by binding it to a name with `<-`.

I/O actions will only be performed when they are given a name of `main` or when they're inside a bigger I/O action that we composed with a *do* block. We can also use a *do* block to glue together a few I/O actions and then we can use that I/O action in another *do* block and so on. Either way, they'll be performed only if they eventually fall into `main`.

Oh, right, there's also one more case when I/O actions will be performed. When we type out an I/O action in GHCI and press return, it will be performed.

```
ghci> putStrLn "HEEY"
HEEY
```

Even when we just punch out a number or call a function in GHCI and press return, it will evaluate it (as much as it needs) and then call `show` on it and then it will print that string to the terminal using `putStrLn` implicitly.

Remember *let* bindings? If you don't, refresh your memory on them by reading this section. They have to be in the form of `let <bindings> in <expression>`, where `<bindings>` are names to be given to expressions and `<expression>` is the expression that is to be evaluated that sees them. We also said that in list comprehensions, the *in* part isn't needed. Well, you can use them in *do* blocks pretty much like you use them in list comprehensions. Check this out:

```
import Data.Char
main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are
you?"
```

See how the I/O actions in the *do* block are lined up? Also notice how the *let* is lined up with the I/O actions and the names of the *let* are lined up with each other? That's good practice, because indentation is important in Haskell. Now, we did `map toUpper firstName`, which turns something like `"John"` into a much cooler string like `"JOHN"`. We bound that uppercased string to a name and then used it in a string later on that we printed to the terminal.

You may be wondering when to use `<-` and when to use *let* bindings? Well, remember, `<-` is (for now) for performing I/O actions and binding their results to names. `map toUpper firstName`, however, isn't an I/O action. It's a pure expression in Haskell. So use `<-` when you want to bind results of I/O actions to names and you can use *let* bindings to bind pure expressions to names. Had we done something like `let firstName = getLine`, we would have just called the `getLine` I/O action a different name and we'd still have to run it through a `<-` to perform it.

Now we're going to make a program that continuously reads a line and prints out the same line with the words reversed. The program's execution

will stop when we input a blank line. This is the program:

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

To get a feel of what it does, you can run it before we go over the code.

> **Protip**: To run a program you can either compile it and then run the produced executable file by doing `ghc --make helloworld` and then `./helloworld` or you can use the `runhaskell` command like so: `runhaskell helloworld.hs` and your program will be executed on the fly.

First, let's take a look at the `reverseWords` function. It's just a normal function that takes a string like `"hey there man"` and then calls `words` with it to produce a list of words like `["hey","there","man"]`. Then we map `reverse` on the list, getting `["yeh","ereht","nam"]` and then we put that back into one string by using `unwords` and the final result is `"yeh ereht nam"`. See how we used function composition here. Without function composition, we'd have to write something like `reverseWords st = unwords (map reverse (words st))`.

What about `main`? First, we get a line from the terminal by performing `getLine` call that line `line`. And now, we have a conditional expression. Remember that in Haskell, every *if* must have a corresponding *else* because every expression has to have some sort of value. We make the *if* so that when a condition is true (in our case, the line that we entered is blank), we run perform one I/O action and when it isn't, the I/O action under the *else* is performed. That's why in an I/O *do* block, *if*s have to have a form of `if <condition> then <I/O action> else <I/O action>`.

Let's first take a look at what happens under the *else* clause. Because, we have to have exactly one I/O action after the *else*, we use a *do* block to glue together two I/O actions into one. You could also write that part out as:

```
        else (do
            putStrLn $ reverseWords line
            main)
```

This makes it more explicit that the *do* block can be viewed as one I/O action, but it's uglier. Anyway, inside the *do* block, we call `reverseWords` on the line that we got from `getLine` and then print that out to the terminal. After that, we just perform `main`. It's called recursively and that's okay, because `main` is itself an I/O action. So in a sense, we go back to the start of the program.

Now what happens when `null line` holds true? What's after the *then* is performed in that case. If we look up, we'll see that it says `then return ()`. If you've done imperative languages like C, Java or Python, you're probably thinking that you know what this `return` does and chances are you've already

skipped this really long paragraph. Well, here's the thing: **the `return` in Haskell is really nothing like the `return` in most other languages!** It has the same name, which confuses a lot of people, but in reality it's quite different. In imperative languages, `return` usually ends the execution of a method or subroutine and makes it report some sort of value to whoever called it. In Haskell (in I/O actions specifically), it makes an I/O action out of a pure value. If you think about the box analogy from before, it takes a value and wraps it up in a box. The resulting I/O action doesn't actually do anything, it just has that value encapsulated as its result. So in an I/O context, `return "haha"` will have a type of `IO String`. What's the point of just transforming a pure value into an I/O action that doesn't do anything? Why taint our program with `IO` more than it has to be? Well, we needed some I/O action to carry out in the case of an empty input line. That's why we just made a bogus I/O action that doesn't do anything by writing `return ()`.

Using `return` doesn't cause the I/O *do* block to end in execution or anything like that. For instance, this program will quite happily carry out all the way to the last line:

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

All these `return`s do is that they make I/O actions that don't really do anything except have an encapsulated result and that result is thrown away because it isn't bound to a name. We can use `return` in combination with `<-` to bind stuff to names.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

So you see, `return` is sort of the opposite to `<-`. While `return` takes a value and wraps it up in a box, `<-` takes a box (and performs it) and takes the value out of it, binding it to a name. But doing this is kind of redundant, especially since you can use *let* bindings in *do* blocks to bind to names, like so:

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

When dealing with I/O *do* blocks, we mostly use `return` either because we need to create an I/O action that doesn't do anything or because we don't want the I/O action that's made up from a *do* block to have the result value of its last action, but we want it to have a different result value, so we use `return` to make an I/O action that always has our desired result contained and we put it at the end.

A *do* block can also have just one I/O action. In that case, it's the same as just writing the I/O action. Some people would prefer writing `then do return ()` in this case because the *else* also has a *do*.

Before we move on to files, let's take a look at some functions that are useful when dealing with I/O.

`putStr` is much like `putStrLn` in that it takes a string as a parameter and returns an I/O action that will print that string to the terminal, only `putStr` doesn't jump into a new line after printing out the string while `putStrLn` does.

```
main = do   putStr "Hey, "
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

Its type signature is `putStr :: String -> IO ()`, so the result encapsulated within the resulting I/O action is the unit. A dud value, so it doesn't make sense to bind it.

`putChar` takes a character and returns an I/O action that will print it out to the terminal.

```
main = do   putChar 't'
            putChar 'e'
            putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`putStr` is actually defined recursively with the help of `putChar`. The edge condition of `putStr` is the empty string, so if we're printing an empty string, just return an I/O action that does nothing by using `return ()`. If it's not empty, then print the first character of the string by doing `putChar` and then print of them using `putStr`.

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

See how we can use recursion in I/O, just like we can use it in pure code. Just like in pure code, we define the edge case and then think what the result actually is. It's an action that first outputs the first character and then outputs the rest of the string.

`print` takes a value of any type that's an instance of `Show` (meaning that we know how to represent it as a string), calls `show` with that value to stringify it and then outputs that string to the terminal. Basically, it's just `putStrLn . show`. It first runs `show` on a value and then feeds that to `putStrLn`, which returns an I/O action that will print out our value.

```
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

```
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

As you can see, it's a very handy function. Remember how we talked about how I/O actions are performed only when they fall into `main` or when we try to evaluate them in the GHCI prompt? When we type out a value (like `3` or `[1,2,3]`) and press the return key, GHCI actually uses `print` on that value to display it on our terminal!

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print (map (++"!") ["hey","ho","woo"])
["hey!","ho!","woo!"]
```

When we want to print out strings, we usually use `putStrLn` because we don't want the quotes around them, but for printing out values of other types to the terminal, `print` is used the most.

`getChar` is an I/O action that reads a character from the input. Thus, its type signature is `getChar :: IO Char`, because the result contained within the I/O action is a `Char`. Note that due to buffering, reading of the characters won't actually happen until the user mashes the return key.

```
main = do
    c <- getChar
    if c /= ' '
        then do
            putChar c
            main
        else return ()
```

This program looks like it should read a character and then check if it's a space. If it is, halt execution and if it isn't, print it to the terminal and then do the same thing all over again. Well, it kind of does, only not in the way you might expect. Check this out:

```
$ runhaskell getchar_test.hs
hello sir
hello
```

The second line is the input. We input `hello sir` and then press return. Due to buffering, the execution of the program will begin only when after we've hit return and not after every inputted character. But once we press return, it acts on what we've been putting in so far. Try playing with this program to get a feel for it!

The `when` function is found in `Control.Monad` (to get access to it, do `import Control.Monad`). It's interesting because in a *do* block it looks like a control flow statement, but it's actually a normal function. It takes a boolean value and an I/O action if that boolean value is `True`, it returns the same I/O action that we supplied to

it. However, if it's `False`, it returns the `return ()`, action, so an I/O action that doesn't do anything. Here's how we could rewrite the previous piece of code with which we demonstrated `getChar` by using `when`:

```
import Control.Monad
main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

So as you can see, it's useful for encapsulating the `if <something> then do <some I/O action> else return`
pattern.

`sequence` takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other. The result contained in that I/O action will be a list of the results of all the I/O actions that were performed. Its type signature is `sequence :: [IO a] -> IO [a]`. Doing this:

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

Is exactly the same as doing this:.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

So `sequence [getLine, getLine, getLine]` makes an I/O action that will perform `getLine` three times. If we bind that action to a name, the result is a list of all the results, so in our case, a list of three things that the user entered at the prompt.

A common pattern with `sequence` is when we map functions like `print` or `putStrLn` over lists. Doing `map print [1,2,3,4]` won't create an I/O action. It will create a list of I/O actions, because that's like writing `[print 1, print 2, print 3, print 4]`. If we want to transform that list of I/O actions into an I/O action, we have to sequence it.

```
ghci> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

What's with the `[(),(),(),(),()]` at the end? Well, when we evaluate an I/O action in GHCI, it's performed and then its result is printed out, unless that result is `()`, in which case it's not printed out. That's why evaluating `putStrLn "hehe"` in GHCI just prints out `hehe` (because the contained result in `putStrLn "hehe"` is `()`). But when we do `getLine` in GHCI, the result of that I/O action is printed out, because `getLine` has a type of `IO String`.

Because the mapping a function that returns an I/O action over a list and then sequencing it so common, the utility functions `mapM` and `mapM_` were

introduced. `mapM` takes a function and a list, maps the function over the list
and then sequences it. `mapM_` does the same, only it throws away the result
later. We usually use `mapM_` when we don't care what result our sequenced I/O
actions have.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` takes an I/O action and returns an I/O action that just repeats the
I/O action it got forever. It's located in `Control.Monad`. This little program will
indefinitely ask the user for some input and spit it back to him, CAPSLOCKED:

```
import Control.Monad
import Data.Char
main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forM` (located in `Control.Monad`) is like `mapM`, only that it has its parameters
switched around. The first parameter is the list and the second one is the
function to map over that list, which is then sequenced. Why is that useful?
Well, with some creative use of lambdas and *do* notation, we can do stuff like
this:

```
import Control.Monad
main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a
++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

The (`a-> do ...`) is a function that takes a number and returns an I/O
action. We have to surround it with parentheses, otherwise the lambda
thinks the last two I/O actions belong to it. Notice that we do `return color`
in the inside *do* block. We do that so that the I/O action which the *do* block
defines has the result of our color contained within it. We actually didn't
have to do that, because `getLine` already has that contained within it. Doing
`color <- getLine` and then `return color` is just unpacking the result from `getLine`
and then repackaging it again, so it's the same as just doing `getLine`. The `forM`
(called with its two parameters) produces an I/O action, whose result we bind
to `colors`. `colors` is just a normal list that holds strings. At the end, we print
out all those colors by doing `mapM putStrLn colors`.

You can think of `forM` as meaning: make an I/O action for every element
in this list. What each I/O action will do can depend on the element that was
used to make the action. Finally, perform those actions and bind their results
to something. We don't have to bind it, we can also just throw it away.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

We could have actually done that without `forM`, only with `forM` it's more readable. Normally we write `forM` when we want to map and sequence some actions that we define there on the spot using *do* notation. In the same vein, we could have replaced the last line with `forM colors putStrLn`.

In this section, we learned the basics of input and output. We also found out what I/O actions are, how they enable us to do input and output and when they are actually performed. To reiterate, I/O actions are values much like any other value in Haskell. We can pass them as parameters to functions and functions can return I/O actions as results. What's special about them is that if they fall into the `main` function (or are the result in a GHCI line), they are performed. And that's when they get to write stuff on your screen or play Yakety Sax through your speakers. Each I/O action can also encapsulate a result with which it tells you what it got from the real world.

Don't think of a function like `putStrLn` as a function that takes a string and prints it to the screen. Think of it as a function that takes a string and returns an I/O action. That I/O action will, when performed, print beautiful poetry to your terminal.

9.2   Files and streams

`getChar` is an I/O action that reads a single character from the terminal. `getLine` is an I/O action that reads a line from the terminal. These two are pretty straightforward and most programming languages have some functions or statements that are parallel to them. But now, let's meet `getContents`. getContents is an I/O action that reads everything from the standard input until it encounters an end-of-file character. Its type is `getContents :: IO String`. What's cool about `getContents` is that it does lazy I/O. When we do `foo <- getContents`, it doesn't read all of the input at once, store it in memory and then bind it to `foo`. No, it's lazy! It'll say: *"Yeah yeah, I'll read the input from the terminal later as we go along, when you really need it!"*.

`getContents` is really useful when we're piping the output from one program into the input of our program. In case you don't know how piping works in unix-y systems, here's a quick primer. Let's make a text file that contains the following little haiku:

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Yeah, the haiku sucks, what of it? If anyone knows of any good haiku tutorials, let me know.

Now, recall the little program we wrote when we were introducing the `forever` function. It prompted the user for a line, returned it to him in CAP-SLOCK and then did that all over again, indefinitely. Just so you don't have to scroll all the way back, here it is again:

```
import Control.Monad
import Data.Char
main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

We'll save that program as `capslocker.hs` or something and compile it. And then, we're going to use a unix pipe to feed our text file directly to our little program. We're going to use the help of the GNU *cat* program, which prints out a file that's given to it as an argument. Check it out, booyaka!

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ cat haiku.txt
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

As you can see, piping the output of one program (in our case that was *cat*) to the input of another (*capslocker*) is done with the | character. What we've done is pretty much equivalent to just running *capslocker*, typing our haiku at the terminal and then issuing an end-of-file character (that's usually done by pressing Ctrl-D). It's like running *cat haiku.txt* and saying: "Wait, don't print this out to the terminal, tell it to *capslocker* instead!".

So what we're essentially doing with that use of `forever` is taking the input and transforming it into some output. That's why we can use `getContents` to make our program even shorter and better:

```
import Data.Char
main = do
    contents <- getContents
    putStr (map toUpper contents)
```

We run the `getContents` I/O action and name the string it produces `contents`. Then, we map `toUpper` over that string and print that to the terminal. Keep in mind that because strings are basically lists, which are lazy, and `getContents` is I/O lazy, it won't try to read the whole content at once and store it into memory before printing out the capslocked version. Rather, it will print out

the capslocked version as it reads it, because it will only read a line from the
input when it really needs to.

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

Cool, it works. What if we just run *capslocker* and try to type in the lines
ourselves?

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

We got out of that by pressing Ctrl-D. Pretty nice! As you can see, it
prints out our capslocked input back to us line by line. When the result of
getContents is bound to contents, it's not represented in memory as a real string,
but more like a promise that it will produce the string eventually. When we
map toUpper over contents, that's also a promise to map that function over the
eventual contents. And finally when putStr happens, it says to the previous
promise: *"Hey, I need a capslocked line!"*. It doesn't have any lines yet, so it
says to contents: *"Hey, how about actually getting a line from the terminal?"*. So
that's when getContents actually reads from the terminal and gives a line to
the code that asked it to produce something tangible. That code then maps
toUpper over that line and gives it to putStr, which prints it. And then, putStr
says: *"Hey, I need the next line, come on!"* and this repeats until there's no more
input, which is signified by an end-of-file character.

Let's make program that takes some input and prints out only those lines
that are shorter than 10 characters. Observe:

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)
shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

We've made our I/O part of the program as short as possible. Because our
program is supposed to take some input and print out some output based
on the input, we can implement it by reading the input contents, running a
function on them and then printing out what the function gave back.

The shortLinesOnly function works like this: it takes a string, like "short\nlooooooooooooooooong\nshort aga
That string has three lines, two of them are short and the middle one
is long. It runs the lines function on that string, which converts it to
["short", "looooooooooooooong", "short again"], which we then bind to the name
allLines. That list of string is then filtered so that only those lines that are
shorter than 10 characters remain in the list, producing ["short", "short again"].
And finally, unlines joins that list into a single newline delimited string, giving
"short\nshort again". Let's give it a go.

```
i'm short
so am i
i am a loooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooooong
short
```

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
short
```

We pipe the contents of *shortlines.txt* into the output of *shortlinesonly* and as the output, we only get the short lines.

This pattern of getting some string from the input, transforming it with a function and then outputting that is so common that there exists a function which makes that even easier, called `interact`. `interact` takes a function of type `String -> String` as a parameter and returns an I/O action that will take some input, run that function on it and then print out the function's result. Let's modify our program to use that.

```
main = interact shortLinesOnly
shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Just to show that this can be achieved in much less code (even though it will be less readable) and to demonstrate our function composition skill, we're going to rework that a bit further.

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Wow, we actually reduced that to just one line, which is pretty cool!

`interact` can be used to make programs that are piped some contents into them and then dump some result out or it can be used to make programs that appear to take a line of input from the user, give back some result based on that line and then take another line and so on. There isn't actually a real distinction between the two, it just depends on how the user is supposed to use them.

Let's make a program that continuously reads a line and then tells us if the line is a palindrome or not. We could just use `getLine` to read a line, tell the user if it's a palindrome and then run `main` all over again. But it's simpler if we use `interact`. When using `interact`, think about what you need to do to transform some input into the desired output. In our case, we have to replace each line of the input with either `"palindrome"` or `"not a palindrome"`. So we have to write a function that transforms something like `"elephant\nABCBA\nwhatever"` into `"not a palindrome\npalindrome\nnot a palindrome"`. Let's do this!

```
respondPalindromes contents = unlines (map (\xs -> if isPalindrome xs then
"palindrome" else "not a palindrome") (lines contents))
    where   isPalindrome xs = xs == reverse xs
```

Let's write this in point-free.

```
respondPalindromes = unlines . map (\xs -> if isPalindrome xs then
"palindrome" else "not a palindrome") . lines
    where   isPalindrome xs = xs == reverse xs
```

Pretty straightforward. First it turns something like `"elephant\nABCBA\nwhatever"` into `["elephant", "ABCBA", "whatever"]` and then it maps that lambda over it, giving `["not a palindrome", "palindrome", "not a palindrome"]` and then `unlines` joins that list into a single, newline delimited string. Now we can do

```
main = interact respondPalindromes
```

Let's test this out:

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

Even though we made a program that transforms one big string of input into another, it acts like we made a program that does it line by line. That's because Haskell is lazy and it wants to print the first line of the result string, but it can't because it doesn't have the first line of the input yet. So as soon as we give it the first line of input, it prints the first line of the output. We get out of the program by issuing an end-of-line character.

We can also use this program by just piping a file into it. Let's say we have this file:

```
dogaroo
radar
rotor
madam
```

and we save it as `words.txt`. This is what we get by piping it into our program:

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

Again, we get the same output as if we had run our program and put in the words ourselves at the standard input. We just don't see the input that `palindromes.hs` because the input came from the file and not from us typing the words in.

So now you probably see how lazy I/O works and how we can use it to our advantage. You can just think in terms of what the output is supposed to be for some given input and write a function to do that transformation. In lazy I/O, nothing is eaten from the input until it absolutely has to be because

what we want to print right now depends on that input.

So far, we've worked with I/O by printing out stuff to the terminal and reading from it. But what about reading and writing files? Well, in a way, we've already been doing that. One way to think about reading from the terminal is to imagine that it's like reading from a (somewhat special) file. Same goes for writing to the terminal, it's kind of like writing to a file. We can call these two files `stdout` and `stdin`, meaning *standard output* and *standard input*, respectively. Keeping that in mind, we'll see that writing to and reading from files is very much like writing to the standard output and reading from the standard input.

We'll start off with a really simple program that opens a file called *girl-friend.txt*, which contains a verse from Avril Lavigne's #1 hit *Girlfriend*, and just prints out out to the terminal. Here's *girlfriend.txt*:

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

And here's our program:

```
import System.IO
main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Running it, we get the expected result:

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Let's go over this line by line. The first line is just four exclamations, to get our attention. In the second line, Avril tells us that she doesn't like our current romantic partner. The third line serves to emphasize that disapproval, whereas the fourth line suggests we should seek out a new girlfriend.

Let's also go over the program line by line! Our program is several I/O actions glued together with a *do* block. In the first line of the *do* block, we notice a new function called `openFile`. This is its type signature: `openFile :: FilePath -> IOMode -> IO Handle`. If you read that out loud, it states: `openFile` takes a file path and an `IOMode` and returns an I/O action that will open a file and have the file's associated handle encapsulated as its result.

`FilePath` is just a type synonym for `String`, simply defined as:

```
type FilePath = String
```

`IOMode` is a type that's defined like this:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Just like our type that represents the seven possible values for the days of the week, this type is an enumeration that represents what we want to do with our opened file. Very simple. Just note that this type is `IOMode` and not `IO Mode`. `IO Mode` would be the type of an I/O action that has a value of some type `Mode` as its result, but `IOMode` is just a simple enumeration.

Finally, it returns an I/O action that will open the specified file in the specified mode. If we bind that action to something we get a `Handle`. A value of type `Handle` represents where our file is. We'll use that handle so we know which file to read from. It would be stupid to read a file but not bind that read to a handle because we wouldn't be able to do anything with the file. So in our case, we bound the handle to `handle`.

In the next line, we see a function called `hGetContents`. It takes a `Handle`, so it knows which file to get the contents from and returns an `IO String` — an I/O action that holds as its result the contents of the file. This function is pretty much like `getContents`. The only difference is that `getContents` will automatically read from the standard input (that is from the terminal), whereas `hGetContents` takes a file handle which tells it which file to read from. In all other respects, they work the same. And just like `getContents`, `hGetContents` won't attempt to read the file at once and store it in memory, but it will read it as needed. That's really cool because we can treat `contents` as the whole contents of the file, but it's not really loaded in memory. So if this were a really huge file, doing `hGetContents` wouldn't choke up our memory, but it would read only what it needed to from the file, when it needed to.

Note the difference between the handle used to identify a file and the contents of the file, bound in our program to `handle` and `contents`. The handle is just something by which we know what our file is. If you imagine your whole file system to be a really big book and each file is a chapter in the book, the handle is a bookmark that shows where you're currently reading (or writing) a chapter, whereas the contents are the actual chapter.

With `putStr contents` we just print the contents out to the standard output and then we do `hClose`, which takes a handle and returns an I/O action that closes the file. You have to close the file yourself after opening it with `openFile`!

Another way of doing what we just did is to use the `withFile` function, which has a type signature of `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`. It takes a path to a file, an `IOMode` and then it takes a function that takes a handle and returns some I/O action. What it returns is an I/O action that will open that file, do something we want with the file and then close it. The result encapsulated in the final I/O action that's returned is the same as the result of the I/O action that the function we give it returns. This might sound a bit complicated, but it's really simple, especially with lambdas, here's our previous example rewritten to use `withFile`:

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
```

```
        contents <- hGetContents handle
        putStr contents)
```

As you can see, it's very similar to the previous piece of code. `(\handle -> ... )` is the function that takes a handle and returns an I/O action and it's usually done like this, with a lambda. The reason it has to take a function that returns an I/O action instead of just taking an I/O action to do and then close the file is because the I/O action that we'd pass to it wouldn't know on which file to operate. This way, `withFile` opens the file and then passes the handle to the function we gave it. It gets an I/O action back from that function and then makes an I/O action that's just like it, only it closes the file afterwards. Here's how we can make our own `withFile` function:

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

We know the result will be an I/O action so we can just start off with a *do*. First we open the file and get a handle from it. Then, we apply `handle` to our function to get back the I/O action that does all the work. We bind that action to `result`, close the handle and then do `return result`. By `return`ing the result encapsulated in the I/O action that we got from `f`, we make it so that our I/O action encapsulates the same result as the one we got from `f handle`. So if `f handle` returns an action that will read a number of lines from the standard input and write them to a file and have as its result encapsulated the number of lines it read, if we used that with `withFile'`, the resulting I/O action would also have as its result the number of lines read.

Just like we have `hGetContents` that works like `getContents` but for a specific file, there's also `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar`, etc. They work just like their counterparts without the *h*, only they take a handle as a parameter and operate on that specific file instead of operating on standard input or standard output. Example: `putStrLn` is a function that takes a string and returns an I/O action that will print out that string to the terminal and a newline after it. `hPutStrLn` takes a handle and a string and returns an I/O action that will write that string to the file associated with the handle and then put a newline after it. In the same vein, `hGetLine` takes a handle and returns an I/O action that reads a line from its file.

Loading files and then treating their contents as strings is so common that we have these three nice little functions to make our work even easier:

`readFile` has a type signature of `readFile :: FilePath -> IO String`. Remember, `FilePath` is just a fancy name for `String`. `readFile` takes a path to a file and returns an I/O action that will read that file (lazily, of course) and bind its contents to something as a string. It's usually more handy than doing `openFile` and binding it to a handle and then doing `hGetContents`. Here's how we could have written our previous example with `readFile`:

```
import System.IO
main = do
```

```
    contents <- readFile "girlfriend.txt"
    putStr contents
```

Because we don't get a handle with which to identify our file, we can't close it manually, so Haskell does that for us when we use `readFile`.

`writeFile` has a type of `writeFile :: FilePath -> String -> IO ()`. It takes a path to a file and a string to write to that file and returns an I/O action that will do the writing. If such a file already exists, it will be stomped down to zero length before being written on. Here's how to turn *girlfriend.txt* into a CAPSLOCKED version and write it to *girlfriendcaps.txt*:

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "girlfriendcaps.txt" (map toUpper contents)
```

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

`appendFile` has a type signature that's just like `writeFile`, only `appendFile` doesn't truncate the file to zero length if it already exists but it appends stuff to it.

Let's say we have a file *todo.txt* that has one task per line that we have to do. Now let's make a program that takes a line from the standard input and adds that to our to-do list.

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

We needed to add the `"n"` to the end of each line because `getLine` doesn't give us a newline character at the end.

Ooh, one more thing. We talked about how doing `contents <- hGetContents handle` doesn't cause the whole file to be read at once and stored in-memory. It's I/O lazy, so doing this:

```
main = do
    withFile "something.txt" ReadMode (\handle ->
        contents <- hGetContents handle
        putStr contents)
```

is actually like connecting a pipe from the file to the output. Just like you can think of lists as streams, you can also think of files as streams. This will read one line at a time and print it out to the terminal as it goes along. So you may be asking, how wide is this pipe then? How often will the disk be accessed? Well, for text files, the default buffering is line-buffering usually. That means that the smallest part of the file to be read at once is one line. That's why in this case it actually reads a line, prints it to the output, reads the next line, prints it, etc. For binary files, the default buffering is usually block-buffering. That means that it will read the file chunk by chunk. The chunk size is some size that your operating system thinks is cool.

You can control how exactly buffering is done by using the `hSetBuffering` function. It takes a handle and a `BufferMode` and returns an I/O action that sets the buffering. `BufferMode` is a simple enumeration data type and the possible values it can hold are: `NoBuffering`, `LineBuffering` or `BlockBuffering (Maybe Int)`. The `Maybe Int` is for how big the chunk should be, in bytes. If it's `Nothing`, then the operating system determines the chunk size. `NoBuffering` means that it will be read one character at a time. `NoBuffering` usually sucks as a buffering mode because it has to access the disk so much.

Here's our previous piece of code, only it doesn't read it line by line but reads the whole file in chunks of 2048 bytes.

```
main = do
    withFile "something.txt" ReadMode (\handle ->
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Reading files in bigger chunks can help if we want to minimize disk access or when our file is actually a slow network resource.

We can also use `hFlush`, which is a function that takes a handle and returns an I/O action that will flush the buffer of the file associated with the handle. When we're doing line-buffering, the buffer is flushed after every line. When we're doing block-buffering, it's after we've read a chunk. It's also flushed after closing a handle. That means that when we've reached a newline character, the reading (or writing) mechanism reports all the data so far. But we can use `hFlush` to force that reporting of data that has been read so far. After flushing, the data is available to other programs that are running at the same time.

Think of reading a block-buffered file like this: your toilet bowl is set to flush itself after it has one gallon of water inside it. So you start pouring in water and once the gallon mark is reached, that water is automatically flushed and the data in the water that you've poured in so far is read. But you can flush the toilet manually too by pressing the button on the toilet. This makes the toilet flush and all the water (data) inside the toilet is read. In case you haven't noticed, flushing the toilet manually is a metaphor for `hFlush`. This is

not a very great analogy by programming analogy standards, but I wanted a real world object that can be flushed for the punchline.

We already made a program to add a new item to our to-do list in *todo.txt*, now let's make a program to remove an item. I'll just paste the code and then we'll go over the program together so you see that it's really easy. We'll be using a few new functions from `System.Directory` and one new function from `System.IO`, but they'll all be explained.

Anyway, here's the program for removing an item from *todo.txt*:

```
import System.IO
import System.Directory
import Data.List
main = do
    handle <- openFile "todo.txt" ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..]
todoTasks
    putStrLn "These are your TO-DO items:"
    putStr $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

At first, we just open *todo.txt* in read mode and get bind its handle to `handle`. Then, we use an I/O action from `System.Directory` called $\boxed{\texttt{getTemporaryDirectory}}$. This is an I/O action that asks your operating system what a suitable directory for storing a temporary file is and reports that as its result. Its type is `getTemporaryDirectory :: IO String`, although it would be clearer in this case to look at it as `getTemporaryDirectory :: IO FilePath`. We bind the file path of the temporary directory to `tempdir`.

Next up, we use a function that we haven't met before which is from `System.IO` — $\boxed{\texttt{openTempFile}}$. Its name is pretty self-explanatory. It takes a path to a temporary directory and a template name for a file and opens a temporary file. We used `"temp"` as the template name for the temporary file, which means that the temporary file will be named *temp* plus some random characters. It returns an I/O action that makes the temporary file and the result in that I/O action is a pair of values: the name of the temporary file and a handle. We could just open a normal file called *todo2.txt* or something like that but it's better practice to use `getTemporaryDirectory` and `openTempFile` so you know you're not overwriting anything.

Next up, we bind the contents of *todo.txt* to `contents`. Then, split that string into a list of strings, each string one line. So `todoTasks` is now some-

thing like `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]`. We zip the numbers from 0 onwards and that list with a function that takes a number, like 3, and a string, like `"hey"` and returns `"3 - hey"`, so numberedTasks is `["0 - Iron the dishes", "1 - Dust the dog"...`. We join that list of strings into a single newline delimited string with `unlines` and print that string out to the terminal. Note that instead of doing that, we could have also done `mapM putStrLn numberedTasks`.

We ask the user which one they want to delete and wait for them to enter a number. Let's say they want to delete number 1, which is `Dust the dog`, so they punch in `1`. numberString is now `"1"` and because we want a number, not a string, we run `read` on that to get `1` and bind that to `number`.

Remember the `delete` and `!!` functions from `Data.List`. `!!` returns an element from a list with some index and `delete` deletes the first occurence of an element in a list and returns a new list without that occurence. `(todoTasks !! number)` (number is now `1`) returns `"Dust the dog"`. We bind `todoTasks` without the first occurence of `"Dust the dog"` to `newTodoItems` and then join that into a single string with `unlines` before writing it to the temporary file that we opened. The old file is now unchanged and the temporary file contains all the lines that the old one does, except the one we deleted.

After that we close both the original and the temporary files and then we remove the original one with `removeFile`, which, as you can see, takes a path to a file and deletes it. After deleting the old *todo.txt*, we use `renameFile` to rename the temporary file to *todo.txt*. Be careful, `removeFile` and `renameFile` (which are both in `System.Directory` by the way) take file paths as their parameters, not handles.

And that's that! We could have done this in even fewer lines, but we were very careful not to overwrite any existing files and politely asked the operating system to tell us where we can put our temporary file. Let's give this a go!

```
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1
$ cat todo.txt
Iron the dishes
Take salad out of the oven
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0
$ cat todo.txt
Take salad out of the oven
```

9.3   Command line arguments

Dealing with command line arguments is pretty much a necessity if you want to make a script or application that runs on a terminal. Luckily, Haskell's standard library has a nice way of getting command line arguments of a program.

In the previous section, we made one program for adding a to-do item to our to-do list and one program for removing an item. There are two problems with the approach we took. The first one is that we just hardcoded the name of our to-do file in our code. We just decided that the file will be named *todo.txt* and that the user will never have a need for managing several to-do lists.

One way to solve that is to always ask the user which file they want to use as their to-do list. We used that approach when we wanted to know which item the user wants to delete. It works, but it's not so good, because it requires the user to run the program, wait for the program to ask something and then tell that to the program. That's called an interactive program and the difficult bit with interactive command line programs is this — what if you want to automate the execution of that program, like with a batch script? It's harder to make a batch script that interacts with a program than a batch script that just calls one program or several of them.

That's why it's sometimes better to have the user tell the program what they want when they run the program, instead of having the program ask the user once it's run. And what better way to have the user tell the program what they want it to do when they run it than via command line arguments!

The `System.Environment` module has two cool I/O actions. One is `getArgs`, which has a type of `getArgs :: IO [String]` and is an I/O action that will get the arguments that the program was run with and have as its contained result a list with the arguments. `getProgName` has a type of `getProgName :: IO String` and is an I/O action that contains the program name.

Here's a small program that demonstrates how these two work:

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn getArgs
   putStrLn "The program name is:"
   putStrLn progName
```

We bind `getArgs` and `progName` to `args` and `progName`. We say `The arguments are:` and then for every argument in `args`, we do `putStrLn`. Finally, we also print out the program name. Let's compile this as `arg-test`.

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
```

```
second
w00t
multi word arg
The program name is:
arg-test
```

Nice. Armed with this knowledge you could create some cool command
line apps. In fact, let's go ahead and make one. In the previous section,
we made a separate program for adding tasks and a separate program for
deleting them. Now, we're going to join that into one program, what it does
will depend on the command line arguments. We're also going to make it so
it can operate on different files, not just *todo.txt*.

We'll call it simply *todo* and it'll be able to do (haha!) three different things:

- View tasks

- Add tasks

- Delete tasks

We're not going to concern ourselves with possible bad input too much
right now.

Our program will be made so that if we want to add the task `Find the magic sword of power`
to the file *todo.txt*, we have to punch in `todo add todo.txt "Find the magic sword of power"`
in our terminal. To view the tasks we'll just do `todo view todo.txt` and to re-
move the task with the index of 2, we'll do `todo remove todo.txt 2`.

We'll start by making a dispatch association list. It's going to be a simple
association list that has command line arguments as keys and functions as their
corresponding values. All these functions will be of type `[String] -> IO ()`.
They're going to take the argument list as a parameter and return an I/O
action that does the viewing, adding, deleting, etc.

```
import System.Environment
import System.Directory
import System.IO
import Data.List
dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]
```

We have yet to define `main`, `add`, `view` and `remove`, so let's start with `main`:

```
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

First, we get the arguments and bind them to `(command:args)`. If you remem-
ber your pattern matching, this means that the first argument will get bound
to `command` and the rest of them will get bound to `args`. If we call our program
like `todo add todo.txt "Spank the monkey"`, `command` will be `"add"` and `args` will be

```
["todo.xt", "Spank the monkey"].
```

In the next line, we look up our command in the dispatch list. Because `"add"` points to `add`, we get `Just add` as a result. We use pattern matching again to extract our function out of the `Maybe`. What happens if our command isn't in the dispatch list? Well then the lookup will return `Nothing`, but we said we won't concern ourselves with failing gracefully too much, so the pattern matching will fail and our program will throw a fit.

Finally, we call our `action` function with the rest of the argument list. That will return an I/O action that either adds an item, displays a list of items or deletes an item and because that action is part of the `main` *do* block, it will get performed. If we follow our concrete example so far and our `action` function is `add`, it will get called with `args` (so `["todo.txt", "Spank the monkey"]`) and return an I/O action that adds `Spank the monkey` to *todo.txt*.

Great! All that's left now is to implement `add`, `view` and `remove`. Let's start with `add`:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

If we call our program like `todo add todo.txt "Spank the monkey"`, the `"add"` will get bound to `command` in the first pattern match in the `main` block, whereas `["todo.txt", "Spank the monkey"]` will get passed to the function that we get from the dispatch list. So, because we're not dealing with bad input right now, we just pattern match against a list with those two elements right away and return an I/O action that appends that line to the end of the file, along with a newline character.

Next, let's implement the list viewing functionality. If we want to view the items in a file, we do `todo view todo.txt`. So in the first pattern match, `command` will be `"view"` and `args` will be `["todo.txt"]`.

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..]
todoTasks
    putStr $ unlines numberedTasks
```

We already did pretty much the same thing in the program that only deleted tasks when we were displaying the tasks so that the user can choose one for deletion, only here we just display the tasks.

And finally, we're going to implement `remove`. It's going to be very similar to the program that only deleted the tasks, so if you don't understand how deleting an item here works, check out the explanation under that program. The main difference is that we're not hardcoding *todo.txt* but getting it as an argument. We're also not prompting the user for the task number to delete, we're getting it as an argument.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
```

```
    handle <- openFile fileName ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

We opened up the file based on `fileName` and opened a temporary file, deleted the line with the index that the user wants to delete, wrote that to the temporary file, removed the original file and renamed the temporary file back to `fileName`.

Here's the whole program at once, in all its glory!

```
import System.Environment
import System.Directory
import System.IO
import Data.List
dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]

main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..]
todoTasks
    putStr $ unlines numberedTasks
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    tempdir <- getTemporaryDirectory
    (tempName, tempHandle) <- openTempFile tempdir "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

To summarize our solution: we made a dispatch association that maps from commands to functions that take some command line arguments and return an I/O action. We see what the command is and based on that we get

the appropriate function from the dispatch list. We call that function with the rest of the command line arguments to get back an I/O action that will do the appropriate thing and then just perform that action!

In other languages, we might have implemented this with a big switch case statement or whatever, but using higher order functions allows us to just tell the dispatch list to give us the appropriate function and then tell that function to give us an I/O action for some command line arguments.

Let's try our app out!

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
$ ./todo add todo.txt "Pick up children from drycleaners"
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners
$ ./todo remove todo.txt 2
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

Another cool thing about this is that it's easy to add extra functionality. Just add an entry in the dispatch association list and implement the corresponding function and you're laughing! As an exercise, you can try implementing a `bump` function that will take a file and a task number and return an I/O action that bumps that task to the top of the to-do list.

You could make this program fail a bit more gracefully in case of bad input (for example, if someone runs `todo UP YOURS HAHAHAHA`) by making an I/O action that just reports there has been an error (say, `errorExit :: IO ()`) and then check for possible erronous input and if there is erronous input, perform the error reporting I/O action. Another way is to use exceptions, which we will meet soon.

9.4   Randomness

Many times while programming, you need to get some random data. Maybe you're making a game where a dice needs to be thrown or you need to generate some test data to test out your program. There are a lot of uses for random data when programming. Well, actually, pseudo-random, because we all know that the only true source of randomness is a monkey on a unicycle with a cheese in one hand and its butt in the other. In this section, we'll take a look at how to make Haskell generate seemingly random data.

In most other programming languages, you have functions that give you back some random number. Each time you call that function, you get back a (hopefully) different random number. How about Haskell? Well, remem-

ber, Haskell is a pure functional language. What that means is that it has referential transparency. What THAT means is that a function, if given the same parameters twice, must produce the same result twice. That's really cool because it allows us to reason differently about programs and it enables us to defer evaluation until we really need it. If I call a function, I can be sure that it won't do any funny stuff before giving me the results. All that matters are its results. However, this makes it a bit tricky for getting random numbers. If I have a function like this:

```
randomNumber :: (Num a) => a
randomNumber = 4
```

It's not very useful as a random number function because it will always return 4, even though I can assure you that the 4 is completely random, because I used a dice to determine it.

How do other languages make seemingly random numbers? Well, they take various info from your computer, like the current time, how much and where you moved your mouse and what kind of noises you made behind your computer and based on that, give a number that looks really random. The combination of those factors (that randomness) is probably different in any given moment in time, so you get a different random number.

Ah. So in Haskell, we can make a random number then if we make a function that takes as its parameter that randomness and based on that returns some number (or other data type).

Enter the `System.Random` module. It has all the functions that satisfy our need for randomness. Let's just dive into one of the functions it exports then, namely `random`. Here's its type: `random :: (RandomGen g, Random a) =>g -> (a, g)`. Whoa! Some new typeclasses in this type declaration up in here! The `RandomGen` typeclass is for types that can act as sources of randomness. The `Random` typeclass is for things that can take on random values. A boolean value can take on a random value, namely `True` or `False`. A number can also take up a plethora of different random values. Can a function take on a random value? I don't think so, probably not! If we try to translate the type declaration of `random` to English, we get something like: it takes a random generator (that's our source of randomness) and returns a random value and a new random generator. Why does it also return a new generator as well as a random value? Well, we'll see in a moment.

To use our `random` function, we have to get our hands on one of those random generators. The `System.Random` module exports a cool type, namely `StdGen` that is an instance of the `RandomGen` typeclass. We can either make a `StdGen` manually or we can tell the system to give us one based on a multitude of sort of random stuff.

To manually make a random generator, use the `mkStdGen` function. It has a type of `mkStdGen :: Int -> StdGen`. It takes an integer and based on that, gives us a random generator. Okay then, let's try using `random` and `mkStdGen` in tandem to get a (hardly random) number.

```
ghci> random (mkStdGen 100)
```

```
<interactive>:1:0:
    Ambiguous type variable 'a' in the constraint:
      'Random a' arising from a use of 'random' at <interactive>:1:0-20
    Probable fix: add a type signature that fixes these type variable(s)
```

What's this? Ah, right, the `random` function can return a value of any type that's part of the `Random` typeclass, so we have to inform Haskell what kind of type we want. Also let's not forget that it returns a random value and a random generator in a pair.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Finally! A number that looks kind of random! The first component of the tuple is our number whereas the second component is a textual representation of our new random generator. What happens if we call `random` with the same random generator again?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Of course. The same result for the same parameters. So let's try giving it a different random generator as a parameter.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Alright, cool, great, a different number. We can use the type annotation to get different types back from that function.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Let's make a function that simulates tossing a coin three times. If `random` didn't return a new generator along with a random value, we'd have to make this function take three random generators as a parameter and then return coin tosses for each of them. But that sounds wrong because if one generator can make a random value of type `Int` (which can take on a load of different values), it should be able to make three coin tosses (which can take on precisely eight combinations). So this is where `random` returning a new generator along with a value really comes in handy.

We'll represent a coin with a simple `Bool`. `True` is tails, `False` is heads.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

We call `random` with the generator we got as a parameter to get a coin and a new generator. Then we call it again, only this time with our new generator, to get the second coin. We do the same for the third coin. Had we called it

with the same generator every time, all the coins would have had the same
value and we'd only be able to get `(False, False, False)` or `(True, True, True)`
as a result.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Notice that we didn't have to do `random gen :: (Bool, StdGen)`. That's because
we already specified that we want booleans in the type declaration of the
function. That's why Haskell can infer that we want a boolean value in this case.

So what if we want to flip four coins? Or five? Well, there's a function
called `randoms` that takes a generator and returns an infinite sequence of values
based on that generator.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Why doesn't `randoms` return a new generator as well as a list? We could
implement the `randoms` function very easily like this:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

A recursive definition. We get a random value and a new generator from
the current generator and then make a list that has the value as its head and
random numbers based on the new generator as its tail. Because we have to
be able to potentially generate an infinite amount of numbers, we can't give
the new random generator back.

We could make a function that generates a finite stream of numbers and a
new generator like this:

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

Again, a recursive definition. We say that if we want 0 numbers, we
just return an empty list and the generator that was given to us. For any
other number of random values, we first get one random number and a new
generator. That will be the head. Then we say that the tail will be *n - 1*
numbers generated with the new generator. Then we return the head and the
rest of the list joined and the final generator that we got from getting the *n - 1*
random numbers.

What if we want a random value in some sort of range? All the random integers so far were outrageously big or small. What if we want to to throw a dice? Well, we use $\boxed{\texttt{randomR}}$ for that purpose. It has a type of `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`, meaning that it's kind of like `random`, only it takes as its first parameter a pair of values that set the lower and upper bounds and the final value produced will be within those bounds.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

There's also $\boxed{\texttt{randomRs}}$, which produces a stream of random values within our defined ranges. Check this out:

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Nice, looks like a super secret password or something.

You may be asking yourself, what does this section have to do with I/O anyway? We haven't done anything concerning I/O so far. Well, so far we've always made our random number generator manually by making it with some arbitrary integer. The problem is, if we do that in our real programs, they will always return the same random numbers, which is no good for us. That's why `System.Random` offers the $\boxed{\texttt{getStdGen}}$ I/O action, which has a type of `IO StdGen`. When your program starts, it asks the system for a good random number generator and stores that in a so called global generator. `getStdGen` fetches you that global random generator when you bind it to something.

Here's a simple program that generates a random string.

```
import System.Random
main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqplrx
```

Be careful though, just performing `getStdGen` twice will ask the system for the same global generator twice. If you do this:

```
import System.Random
main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen2)
```

you will get the same string printed out twice! One way to get two different strings of length 20 is to set up an infinite stream and then take the first 20

characters and print them out in one line and then take the second set of 20
characters and print them out in the second line. For this, we can use the
`splitAt` function from `Data.List`, which splits a list at some index and returns a
tuple that has the first part as the first component and the second part as the
second component.

```
import System.Random
import Data.List
main = do
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

Another way is to use the `newStdGen` action, which splits our current
random generator into two generators. It updates the global random generator
with one of them and encapsulates the other as its result.

```
import System.Random
main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')
```

Not only do we get a new random generator when we bind `newStdGen` to
something, the global one gets updated as well, so if we do `getStdGen` again
and bind it to something, we'll get a generator that's not the same as `gen`.

Here's a little program that will make the user guess which number it's
thinking of.

```
import System.Random
import Control.Monad(when)
main = do
    gen <- getStdGen
    askForNumber gen
askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        askForNumber newGen
```

We make a function `askForNumber`, which takes a random number generator
and returns an I/O action that will prompt the user for a number and tell him
if he guessed it right. In that function, we first generate a random number
and a new generator based on the generator that we got as a parameter and
call them `randNumber` and `newGen`. Let's say that the number generated was `7`.
Then we tell the user to guess which number we're thinking of. We perform
`getLine` and bind its result to `numberString`. When the user enters `7`, `numberString`
becomes `"7"`. Next, we use `when` to check if the string the user entered is an

empty string. If it is, an empty I/O action of `return ()` is performed, which effectively ends the program. If it isn't, the action consisting of that *do* block right there gets performed. We use `read` on `numberString` to convert it to a number, so `number` is now `7`.

> **Excuse me!** If the user gives us some input here that `read` can't read (like `"haha"`), our program will crash with an ugly error message. If you don't want your program to crash on erronous input, use `reads`, which returns an empty list when it fails to read a string. When it succeeds, it returns a singleton list with a tuple that has our desired value as one component and a string with what it didn't consume as the other.

We check if the number that we entered is equal to the one generated randomly and give the user the appropriate message. And then we call `askForNumber` recursively, only this time with the new generator that we got, which gives us an I/O action that's just like the one we performed, only it depends on a different generator and we perform it.

`main` consists of just getting a random generator from the system and calling `askForNumber` with it to get the initial action.

Here's our program in action!

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

Another way to make this same program is like this:

```
import System.Random
import Control.Monad(when)
main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

It's very similar to the previous version, only instead of making a function that takes a generator and then calls itself recursively with the new updated generator, we do all the work in `main`. After telling the user whether they were correct in their guess or not, we update the global generator and then call `main` again. Both approaches are valid but I like the first one more since it does less stuff in `main` and also provides us with a function that we can reuse easily.

9.5 Bytestrings

Lists are a cool and useful data structure. So far, we've used them pretty much everywhere. There are a multitude of functions that operate on them and Haskell's laziness allows us to exchange the for and while loops of other languages for filtering and mapping over lists, because evaluation will only happen once it really needs to, so things like infinite lists (and even infinite lists of infinite lists!) are no problem for us. That's why lists can also be used to represent streams, either when reading from the standard input or when reading from files. We can just open a file and read it as a string, even though it will only be accessed when the need arises.

However, processing files as strings has one drawback: it tends to be slow. As you know, `String` is a type synonym for `[Char]`. `Char`s don't have a fixed size, because it takes several bytes to represent a character from, say, Unicode. Furthemore, lists are really lazy. If you have a list like `[1,2,3,4]`, it will be evaluated only when completely necessary. So the whole list is sort of a promise of a list. Remember that `[1,2,3,4]` is syntactic sugar for `1:2:3:4:[]`. When the first element of the list is forcibly evaluated (say by printing it), the rest of the list `2:3:4:[]` is still just a promise of a list, and so on. So you can think of lists as promises that the next element will be delivered once it really has to and along with it, the promise of the element after it. It doesn't take a big mental leap to conclude that processing a simple list of numbers as a series of promises might not be the most efficient thing in the world.

That overhead doesn't bother us so much most of the time, but it turns out to be a liability when reading big files and manipulating them. That's why Haskell has **bytestrings**. Bytestrings are sort of like lists, only each element is one byte (or 8 bits) in size. The way they handle laziness is also different.

Bytestrings come in two flavors: strict and lazy ones. Strict bytestrings reside in `Data.ByteString` and they do away with the laziness completely. There are no promises involved; a strict bytestring represents a series of bytes in an array. You can't have things like infinite strict bytestrings. If you evaluate the first byte of a strict bytestring, you have to evaluate it whole. The upside is that there's less overhead because there are no thunks (the technical term for *promise*) involved. The downside is that they're likely to fill your memory up faster because they're read into memory at once.

The other variety of bytestrings resides in `Data.ByteString.Lazy`. They're lazy, but not quite as lazy as lists. Like we said before, there are as many thunks in a list as there are elements. That's what makes them kind of slow for some purposes. Lazy bytestrings take a different approach — they are stored in chunks (not to be confused with thunks!), each chunk has a size of 64K. So if you evaluate a byte in a lazy bytestring (by printing it or something), the first 64K will be evaluated. After that, it's just a promise for the rest of the chunks. Lazy bytestrings are kind of like lists of strict bytestrings with a size of 64K. When you process a file with lazy bytestrings, it will be read chunk by chunk. This is cool because it won't cause the memory usage to skyrocket and the 64K probably fits neatly into your CPU's L2 cache.

If you look through the documentation for `Data.ByteString.Lazy`, you'll see that it has a lot of functions that have the same names as the ones from `Data.List`, only the type signatures have `ByteString` instead of `[a]` and `Word8` instead of `a` in them. The functions with the same names mostly act the same as the ones that work on lists. Because the names are the same, we're going to do a qualified import in a script and then load that script into GHCI to play with bytestrings.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` has lazy bytestring types and functions, whereas `S` has strict ones. We'll mostly be using the lazy version.

The function `pack` has the type signature `pack :: [Word8] -> ByteString`. What that means is that it takes a list of bytes of type `Word8` and returns a `ByteString`. You can think of it as taking a list, which is lazy, and making it less lazy, so that it's lazy only at 64K intervals.

What's the deal with that `Word8` type? Well, it's like `Int`, only that it has a much smaller range, namely 0-255. It represents an 8-bit number. And just like `Int`, it's in the `Num` typeclass. For instance, we know that the value `5` is polymorphic in that it can act like any numeral type. Well, it can also take the type of `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

As you can see, you usually don't have to worry about the `Word8` too much, because the type system can makes the numbers choose that type. If you try to use a big number, like `336` as a `Word8`, it will just wrap around to `80`.

We packed only a handful of values into a `ByteString`, so they fit inside one chunk. The `Empty` is like the `[]` for lists.

`unpack` is the inverse function of `pack`. It takes a bytestring and turns it into a list of bytes.

`fromChunks` takes a list of strict bytestrings and converts it to a lazy bytestring. `toChunks` takes a lazy bytestring and converts it to a list of strict ones.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

This is good if you have a lot of small strict bytestrings and you want to process them efficiently without joining them into one big strict bytestring in memory first.

The bytestring version of `:` is called `cons` It takes a byte and a bytestring and puts the byte at the beginning. It's lazy though, so it will make a new

chunk even if the first chunk in the bytestring isn't full. That's why it's better
to use the strict version of `cons`, `cons'` if you're going to be inserting a lot of
bytes at the beginning of a bytestring.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8"
(Chunk "9" (Chunk ":" (Chunk ";" (Chunk "<"
Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

As you can see `empty` makes an empty bytestring. See the difference
between `cons` and `cons'`? With the `foldr`, we started with an empty bytestring
and then went over the list of numbers from the right, adding each number to
the beginning of the bytestring. When we used `cons`, we ended up with one
chunk for every byte, which kind of defeats the purpose.

Otherwise, the bytestring modules have a load of functions that are analo-
gous to those in `Data.List`, including, but not limited to, `head`, `tail`, `init`, `null`,
`length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

It also has functions that have the same name and behave the same as some
functions found in `System.IO`, only `String`s are replaced with `ByteString`s. For in-
stance, the `readFile` function in `System.IO` has a type of `readFile :: FilePath -> IO String`,
while the `readFile` from the bytestring modules has a type of `readFile :: FilePath -> IO ByteString`.
Watch out, if you're using strict bytestrings and you attempt to read a file, it
will read it into memory at once! With lazy bytestrings, it will read it into
neat chunks.

Let's make a simple program that takes two filenames as command-line ar-
guments and copies the first file into the second file. Note that `System.Directory`
already has a function called `copyFile`, but we're going to implement our own
file copying function and program anyway.

```
import System.Environment
import qualified Data.ByteString.Lazy as B
main = do
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2
copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

We make our own function that takes two `FilePath`s (remember, `FilePath` is
just a synonym for `String`) and returns an I/O action that will copy one file into
another using bytestring. In the `main` function, we just get the arguments and
call our function with them to get the I/O action, which is then performed.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Notice that a program that doesn't use bytestrings could look just like
this, the only difference is that we used `B.readFile` and `B.writeFile` instead

of `readFile` and `writeFile`. Many times, you can convert a program that uses normal strings to a program that uses bytestrings by just doing the necessary imports and then putting the qualified module names in front of some functions. Sometimes, you have to convert functions that you wrote to work on strings so that they work on bytestrings, but that's not hard.

Whenever you need better performance in a program that reads a lot of data into strings, give bytestrings a try, chances are you'll get some good performance boosts with very little effort on your part. I usually write programs by using normal strings and then convert them to use bytestrings if the performance is not satisfactory.

Chapter 10

Functionally Solving Problems

In this chapter, we'll take a look at a few interesting problems and how to think functionally to solve them as elegantly as possible. We probably won't be introducing any new concepts, we'll just be flexing our newly acquired Haskell muscles and practicing our coding skills. Each section will present a different problem. First we'll describe the problem, then we'll try and find out what the best (or least worst) way of solving it is.

10.1   Reverse Polish notation calculator

Usually when we write mathematical expressions in school, we write them in an infix manner. For instance, we write `10 - (4 + 3) * 2`. `+`, `*` and `-` are infix operators, just like the infix functions we met in Haskell (`+`, `elem`, etc.). This makes it handy because we, as humans, can parse it easily in our minds by looking at such an expression. The downside to it is that we have to use parentheses to denote precedence.

Reverse Polish notation is another way of writing down mathematical expressions. Initially it looks a bit weird, but it's actually pretty easy to understand and use because there's no need for parentheses and it's very easy to punch into a calculator. While most modern calculators use infix notation, some people still swear by RPN calculators. This is what the previous infix expression looks like in RPN: `10 4 3 + 2 * -`. How do we calculate what the result of that is? Well, think of a stack. You go over the expression from left to right. Every time a number is encountered, push it on to the stack. When we encounter an operator, take the two numbers that are on top of the stack (we also say that we *pop* them), use the operator and those two and then push the resulting number back onto the stack. When you reach the end of the expression, you should be left with a single number if the expression was well-formed and that number represents the result.

Let's go over the expression `10 4 3 + 2 * -` together! First we push `10` on to the stack and the stack is now `10`. The next item is `4`, so we push it to the

stack as well. The stack is now `10`, `4`. We do the same with `3` and the stack is now `10`, `4`, `3`. And now, we encounter an operator, namely `+`! We pop the two top numbers from the stack (so now the stack is just `10`), add those numbers together and push that result to the stack. The stack is now `10`, `7`. We push `2` to the stack, the stack for now is `10`, `7`, `2`. We've encountered an operator again, so let's pop `7` and `2` off the stack, multiply them and push that result to the stack. Multiplying `7` and `2` produces a `14`, so the stack we have now is `10`, `14`. Finally, there's a `-`. We pop `10` and `14` from the stack, subtract `14` from `10` and push that back. The number on the stack is now `-4` and because there are no more numbers or operators in our expression, that's our result!

Now that we know how we'd calculate any RPN expression by hand, let's think about how we could make a Haskell function that takes as its parameter a string that contains a RPN expression, like `"10 4 3 + 2 * -"` and gives us back its result.

What would the type of that function be? We want it to take a string as a parameter and produce a number as its result. So it will probably be something like `solveRPN :: (Num a) =>String -> a`.

> **Protip:** it really helps to first think what the type declaration of a function should be before concerning ourselves with the implementation and then write it down. In Haskell, a function's type declaration tells us a whole lot about the function, due to the very strong type system.

Cool. When implementing a solution to a problem in Haskell, it's also good to think back on how you did it by hand and maybe try to see if you can gain any insight from that. Here we see that we treated every number or operator that was separated by a space as a single item. So it might help us if we start by breaking a string like `"10 4 3 + 2 * -"` into a list of items like `["10","4","3","+","2","*","-"]`.

Next up, what did we do with that list of items in our head? We went over it from left to right and kept a stack as we did that. Does the previous sentence remind you of anything? Remember, in the section about folds, we said that pretty much any function where you traverse a list from left to right or right to left one element by element and build up (accumulate) some result (whether it's a number, a list, a stack, whatever) can be implemented with a fold.

In this case, we're going to use a left fold, because we go over the list from left to right. The accumulator value will be our stack and hence, the result from the fold will also be a stack, only as we've seen, it will only have one item.

One more thing to think about is, well, how are we going to represent the stack? I propose we use a list. Also I propose that we keep the top of our stack at the head of the list. That's because adding to the head (beginning) of a list is much faster than adding to the end of it. So if we have a stack of, say, `10`, `4`, `3`, we'll represent that as the list `[3,4,10]`.

Now we have enough information to roughly sketch our function. It's going to take a string, like, `"10 4 3 + 2 * -"` and break it down into a list of items by using `words` to get `["10","4","3","+","2","*","-"]`. Next, we'll do a left fold over that list and end up with a stack that has a single item, so `[-4]`. We take that single item out of the list and that's our final result!

So here's a sketch of that function:

```
import Data.List
solveRPN :: (Num a) => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where   foldingFunction stack item = ...
```

We take the expression and turn it into a list of items. Then we fold over that list of items with the folding function. Mind the `[]`, which represents the starting accumulator. The accumulator is our stack, so `[]` represents an empty stack, which is what we start with. After getting the final stack with a single item, we call `head` on that list to get the item out and then we apply `read`.

So all that's left now is to implement a folding function that will take a stack, like `[4,10]`, and an item, like `"3"` and return a new stack `[3,4,10]`. If the stack is `[4,10]` and the item `"*"`, then it will have to return `[40]`. But before that, let's turn our function into point-free style because it has a lot of parentheses that are kind of freaking me out:

```
import Data.List
solveRPN :: (Num a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction stack item = ...
```

Ah, there we go. Much better. So, the folding function will take a stack and an item and return a new stack. We'll use pattern matching to get the top items of a stack and to pattern match against operators like `"*"` and `"-"`.

```
solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction xs numberString = read numberString:xs
```

We laid this out as four patterns. The patterns will be tried from top to bottom. First the folding function will see if the current item is `"*"`. If it is, then it will take a list like `[3,4,9,3]` and call its first two elements `x` and `y` respectively. So in this case, `x` would be `3` and `y` would be `4`. `ys` would be `[9,3]`. It will return a list that's just like `ys`, only it has `x` and `y` multiplied as its head. So with this we pop the two topmost numbers off the stack, multiply them and push the result back on to the stack. If the item is not `"*"`, the pattern matching will fall through and `"+"` will be checked, and so on.

If the item is none of the operators, then we assume it's a string that represents a number. If it's a number, we just call `read` on that string to get a number from it and return the previous stack but with that number pushed to the top.

And that's it! Also noticed that we added an extra class constraint of
`Read a` to the function declaration, because we call `read` on our string to get the
number. So this declaration means that the result can be of any type that's
part of the `Num` and `Read` typeclasses (like `Int`, `Float`, etc.).

For the list of items `["2","3","+"]`, our function will start folding from the
left. The intial stack will be `[]`. It will call the folding function with `[]` as the
stack (accumulator) and `"2"` as the item. Because that item is not an operator,
it will be `read` and the added to the beginning of `[]`. So the new stack is now
`[2]` and the folding function will be called with `[2]` as the stack and `["3"]` as
the item, producing a new stack of `[3,2]`. Then, it's called for the third time
with `[3,2]` as the stack and `"+"` as the item. This causes these two numbers to
be popped off the stack, added together and pushed back. The final stack is
`[5]`, which is the number that we return.

Let's play around with our function:

```
ghci> solveRPN "10 4 3 + 2 * -"
-4
ghci> solveRPN "2 3 +"
5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 3 -"
87
```

Cool, it works! One nice thing about this function is that it can be easily
modified to support various other operators. They don't even have to be
binary operators. For instance, we can make an operator `"log"` that just pops
one number off the stack and pushes back its logarithm. We can also make a
ternary operators that pop three numbers off the stack and push back a result
or operators like `"sum"` which pop off all the numbers and push back their
sum.

Let's modify our function to take a few more operators. For simplicity's
sake, we'll change its type declaration so that it returns a number of type
`Float`.

```
import Data.List
solveRPN :: String -> Float
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction (x:y:ys) "/" = (y / x):ys
            foldingFunction (x:y:ys) "^" = (y ** x):ys
            foldingFunction (x:xs) "ln" = log x:xs
            foldingFunction xs "sum" = [sum xs]
            foldingFunction xs numberString = read numberString:xs
```

Wow, great! `/` is division of course and `**` is floating point exponentiation.
With the logarithm operator, we just pattern match against a single element
and the rest of the stack because we only need one element to perform its

natural logarithm. With the sum operator, we just return a stack that has only one element, which is the sum of the stack so far.

```
ghci> solveRPN "2.7 ln"
0.9932518
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

Notice that we can include floating point numbers in our expression because `read` knows how to read them.

```
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

I think that making a function that can calculate arbitrary floating point RPN expressions and has the option to be easily extended in 10 lines is pretty awesome.

One thing to note about this function is that it's not really fault tolerant. When given input that doesn't make sense, it will just crash everything. We'll make a fault tolerant version of this with a type declaration of `solveRPN :: String -> Maybe Float` once we get to know monads (they're not scary, trust me!). We could make one right now, but it would be a bit tedious because it would involve a lot of checking for `Nothing` on every step. If you're feeling up to the challenge though, you can go ahead and try it! Hint: you can use `reads` to see if a read was successful or not.
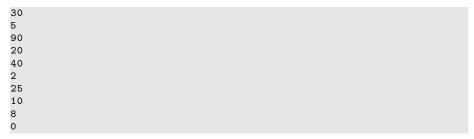
10.2   Heathrow to London

Our next problem is this: your plane has just landed in England and you rent a car. You have a meeting really soon and you have to get from Heathrow Airport to London as fast as you can (but safely!).

There are two main roads going from Heathrow to London and there's a number of regional roads crossing them. It takes you a fixed amount of time to travel from one crossroads to another. It's up to you to find the optimal path to take so that you get to London as fast as you can! You start on the left side and can either cross to the other main road or go forward.

As you can see in the picture, the shortest path from Heathrow to London in this case is to start on main road B, cross over, go forward on A, cross over again and then go forward twice on B. If we take this path, it takes us 75 minutes. Had we chosen any other path, it would take more than that.

Our job is to make a program that takes input that represents a road system and print out what the shortest path across it is. Here's what the input would look like for this case:

```
50
10
```

```
30
5
90
20
40
2
25
10
8
0
```

To mentally parse the input file, read it in threes and mentally split the road system into sections. Each section is comprised of a road A, road B and a crossing road. To have it neatly fit into threes, we say that there's a last crossing section that takes 0 minutes to drive over. That's because we don't care where we arrive in London, as long as we're in London.

Just like we did when solving the RPN calculator problem, we're going to solve this problem in three steps:

- Forget Haskell for a minute and think about how we'd solve the problem by hand

- Think about how we're going to represent our data in Haskell

- Figure out how to operate on that data in Haskell so that we produce at a solution

In the RPN calculator section, we first figured out that when calculating an expression by hand, we'd keep a sort of stack in our minds and then go over the expression one item at a time. We decided to use a list of strings to represent our expression. Finally, we used a left fold to walk oevr the list of strings while keeping a stack to produce a solution.

Okay, so how would we figure out the shortest path from Heathrow to London by hand? Well, we can just sort of look at the whole picture and try to guess what the shortest path is and hopefully we'll make a guess that's right. That solution works for very small inputs, but what if we have a road that has 10,000 sections? Yikes! We also won't be able to say for certain that our solution is the optimal one, we can just sort of say that we're pretty sure.

That's not a good solution then. Here's a simplified picture of our road system:

Alright, can you figure out what the shortest path to the first crossroads (the first blue dot on A, marked *A1*) on road A is? That's pretty trivial. We just see if it's shorter to go directly forward on A or if it's shorter to go forward on B and then cross over. Obviously, it's cheaper to go forward via B and then cross over because that takes 40 minutes, whereas going directly via A takes 50 minutes. What about crossroads *B1*? Same thing. We see that it's a lot cheaper to just go directly via B (incurring a cost of 10 minutes), because going via A and then crossing over would take us a whole 80 minutes!

Now we know what the cheapest path to *A1* is (go via B and then cross over, so we'll say that's B, C with a cost of 40) and we know what the cheapest path to *B1* is (go directly via B, so that's just B, going at 10). Does this knowledge help us at all if we want to know the cheapest path to the next crossroads on both main roads? Gee golly, it sure does!

Let's see what the shortest path to *A2* would be. To get to *A2*, we'll either go directly to *A2* from *A1* or we'll go forward from *B1* and then cross over (remember, we can only move forward or cross to the other side). And because we know the cost to *A1* and *B1*, we can easily figure out what the best path to *A2* is. It costs 40 to get to *A1* and then 5 to get from *A1* to *A2*, so that's B, C, A for a cost of 45. It costs only 10 to get to *B1*, but then it would take an additional 110 minutes to go to *B2* and then cross over! So obviously, the cheapest path to *A2* is B, C, A. In the same way, the cheapest way to *B2* is to go forward from *A1* and then cross over.

> **Maybe you're asking yourself**: but what about getting to *A2* by first crossing over at *B1* and then going on forward? Well, we already covered crossing from *B1* to *A1* when we were looking for the best way to *A1*, so we don't have to take that into account in the next step as well.

Now that we have the best path to *A2* and *B2*, we can repeat this indefinitely until we reach the end. Once we've gotten the best paths for *A4* and *B4*, the one that's cheaper is the optimal path!

So in essence, for the second section, we just repeat the step we did at first, only we take into account what the previous best paths on A and B. We could say that we also took into account the best paths on A and on B in the first step, only they were both empty paths with a cost of 0.

Here's a summary. To get the bast path from Heathrow to London, we do this: first we see what the the best path to the next crossroads on main road A is. The two options are going directly forward or starting at the opposite road, going forward and then crossing over. We remember the cost and the path. We use the same method to see what the best path to the next crossroads on main road B is and remember that. Then, we see if the path to the next crossroads on A is cheaper if we go from the previous A crossroads or if we go from the previous B crossroads and then cross over. We remember the cheaper path and then we do the same for the crossroads opposite of it. We do this for every section until we reach the end. Once we've reached the end, the cheapest of the two paths that we have is our optimal path!

So in essence, we keep one shortest path on the A road and one shortest path on the B road and when we reach the end, the shorter of those two is our path. We now know how to figure out the shortest path by hand. If you had enough time, paper and pencils, you could figure out the shortest path through a road system with any number of sections.

Next step! How do we represent this road system with Haskell's data types? One way is to think of the starting points and crossroads as nodes of a

graph that point to other crossroads. If we imagine that the starting points actually point to each other with a road that has a length of one, we see that every crossroads (or node) points to the node on the other side and also to the next one on its side. Except for the last nodes, they just point to the other side.

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

A node is either a normal node and has information about the road that leads to the other main road and the road that leads to the next node or an end node, which only has information about the road to the other main road. A road keeps information about how long it is and which node it points to. For instance, the first part of the road on the A main road would be `Road 50 a1` where `a1` would be a node `Node x y`, where `x` and `y` are roads that point to *B1* and *A2*.

Another way would be to use `Maybe` for the road parts that point forward. Each node has a road part that point to the opposite road, but only those nodes that aren't the end ones have road parts that point forward.

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

This is an alright way to represent the road system in Haskell and we could certainly solve this problem with it, but maybe we could come up with something simpler? If we think back to our solution by hand, we always just checked the lengths of three road parts at once: the road part on the A road, its opposite part on the B road and part C, which touches those two parts and connects them. When we were looking for the shortest path to *A1* and *B1*, we only had to deal with the lengths of the first three parts, which have lengths of 50, 10 and 30. We'll call that one section. So the road system that we use for this example can be easily represented as four sections: `50, 10, 30, 5, 90, 20, 40, 2, 25,` and `10, 8, 0`.

It's always good to keep our data types as simple as possible, although not any simpler!

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int } deriving
(Show)
type RoadSystem = [Section]
```

This is pretty much perfect! It's as simple as it goes and I have a feeling it'll work perfectly for implementing our solution. `Section` is a simple algebraic data type that holds three integers for the lenghts of its three road parts. We introduce a type synonym as well, saying that `RoadSystem` is a list of sections.

> We could also use a triple of (`Int`, `Int`, `Int`) to represent a road section. Using tuples instead of making your own algebraic data types is good for some small localized stuff, but it's usually better to make a new type for things like this. It gives the type system more information about what's what. We can use (`Int`, `Int`, `Int`) to represent a road section or a vector in 3D space and we can operate on those two, but that allows us to mix them up. If we

use `Section` and `Vector` data types, then we can't accidentally add a
vector to a section of a road system.

Our road system from Heathrow to London can now be represented like
this:

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30, Section 5 90 20, Section 40 2 25,
Section 10 8 0]
```

All we need to do now is to implement the solution that we came up with
previously in Haskell. What should the type declaration for a function that
calculates a shortest path for any given road system be? It should take a road
system as a parameter and return a path. We'll represent a path as a list as
well. Let's introduce a `Label` type that's just an enumeration of either `A`, `B` or `C`.
We'll also make a type synonym: `Path`.

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

Our function, we'll call it `optimalPath` should thus have a type decla-
ration of `optimalPath :: RoadSystem -> Path`. If called with the road system
`heathrowToLondon`, it should return the following path:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

We're going to have to walk over the list with the sections from left to
right and keep the optimal path on A and optimal path on B as we go along.
We'll accumulate the best path as we walk over the list, left to right. What
does that sound like? Ding, ding, ding! That's right, A LEFT FOLD!

When doing the solution by hand, there was a step that we repeated over
and over again. It involed checking the optimal paths on A and B so far and
the current section to produce the new optimal paths on A and B. For instance,
at the beginning the optimal paths were `[]` and `[]` for A and B respectively.
We examined the section `Section 50 10 30` and concluded that the new optimal
path to *A1* is `[(B,10),(C,30)]` and the optimal path to *B1* is `[(B,10)]`. If you look
at this step as a function, it takes a pair of paths and a section and produces a
new pair of paths. The type is `(Path, Path) -> Section -> (Path, Path)`. Let's go
ahead and implement this function, because it's bound to be useful.

> **Hint:** it will be useful because `(Path, Path) -> Section -> (Path, Path)`
> can be used as the binary function for a left fold, which has to
> have a type of `a -> b -> a`

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                        then (A,a):pathA
                        else (C,c):(B,b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
```

```
                          then (B,b):pathB
                          else (C,c):(A,a):pathA
     in  (newPathToA , newPathToB)
```

What's going on here? First, calculate the optimal price on road A based on the best so far on A and we do the same for B. We do `sum $ map snd pathA`, so if `pathA` is something like `[(A,100),(C,20)]`, `priceA` becomes `120`. `forwardPriceToA` is the price that we would pay if we went to the next crossroads on A if we went there directly from the previous crossroads on A. It equals the best price to our previous A, plus the length of the A part of the current section. `crossPriceToA` is the price that we would pay if we went to the next A by going forward from the previous B and then crossing over. It's the best price to the previous B so far plus the B length of the section plus the C length of the section. We determine `forwardPriceToB` and `crossPriceToB` in the same manner.

Now that we know what the best way to A and B is, we just need to make the new paths to A and B based on that. If it's cheaper to go to A by just going forwards, we set `newPathToA` to be `(A,a):pathA`. Basically we prepend the `Label` A and the section length `a` to the optimal path path on A so far. Basically, we say that the best path to the next A crossroads is the path to the previous A crossroads and then one section forward via A. Remember, `A` is just a label, whereas `a` has a type of `Int`. Why do we prepend instead of doing `pathA ++ [(A,a)]`? Well, adding an element to the beginning of a list (also known as consing) is much faster than adding it to the end. This means that the path will be the wrong way around once we fold over a list with this function, but it's easy to reverse the list later. If it's cheaper to get to the next A crossroads by going forward from road B and then crossing over, then `newPathToA` is the old path to B that then goes forward and crosses to A. We do the same thing for `newPathToB`, only everything's mirrored.

Finally, we return `newPathToA` and `newPathToB` in a pair.

Let's run this function on the first section of `heathrowToLondon`. Because it's the first section, the best paths on A and B parameter will be a pair of empty lists.

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

Remember, the paths are reversed, so read them from right to left. From this we can read that the best path to the next A is to start on B and then cross over to A and that the best path to the next B is to just go directly forward from the starting point at B.

> **Optimization tip:** when we do `priceA = sum $ map snd pathA`, we're calculating the price from the path on every step. We wouldn't have to do that if we implemented `roadStep` as a `(Path, Path, Int, Int) -> Section -> (Path, Path, Int, I` function where the integers represent the best price on A and B.

Now that we have a function that takes a pair of paths and a section and produces a new optimal path, we can just easily do a left fold over a list of sections. `roadStep` is called with `([],[])` and the first section and returns a pair of optimal paths to that section. Then, it's called with that pair of paths and

the next section and so on. When we've walked over all the sections, we're left with a pair of optimal paths and the shorter of them is our answer. With this in mind, we can implement `optimalPath`.

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in  if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

We left fold over `roadSystem` (remember, it's a list of sections) with the starting accumulator being a pair of empty paths. The result of that fold is a pair of paths, so we pattern match on the pair to get the paths themselves. Then, we check which one of these was cheaper and return it. Before returning it, we also reverse it, because the optimal paths so far were reversed due to us choosing consing over appending.

Let's test this!

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

This is the result that we were supposed to get! Awesome! It differs from our expected result a bit because there's a step `(C,0)` at the end, which means that we cross over to the other road once we're in London, but because that crossing doesn't cost anything, this is still the correct result.

We have the function that finds an optimal path based on, now we just have to read a textual representation of a road system from the standard input, convert it into a type of `RoadSystem`, run that through our `optimalPath` function and print the path.

First off, let's make a function that takes a list and splits it into groups of the same size. We'll call it `groupsOf`. For a parameter of `[1..10]`, `groupsOf 3` should return `[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

A standard recursive function. For an `xs` of `[1..10]` and an `n` of `3`, this equals `[1,2,3]` : `groupsOf 3 [4,5,6,7,8,9,10]`. When the recursion is done, we get our list in groups of three. And here's our `main` function, which reads from the standard input, makes a `RoadSystem` out of it and prints out the shortest path:

```
import Data.List
main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

First, we get all the contents from the standard input. Then, we call `lines` with our contents to convert something like `"50n10n30n...` to `["50","10","30"..` and then we map `read` to that to convert it to a list of numbers. We call `groupsOf 3` on it so that we turn it to a list of lists of length 3. We map the lambda (`\[a,b,c] -> Section a b c`) over that list of lists. As you can see, the lambda just takes a list of length 3 and turns it into a section. So `roadSystem` is now our system of roads and it even has the correct type, namely `RoadSystem` (or `[Section]`). We call `optimalPath` with that and then get the path and the price in a nice textual representation and print it out.

We save the following text

```
50
10
30
5
90
20
40
2
25
10
8
0
```

in a file called `paths.txt` and then feed it to our program.

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

Works like a charm! You can use your knowledge of the `Data.Random` module to generate a much longer system of roads, which you can then feed to what we just wrote. If you get stack overflows, try using `foldl'` instead of `foldl`, because `foldl'` is strict.

More LYAH coming soon, keeps your eyes peeled!