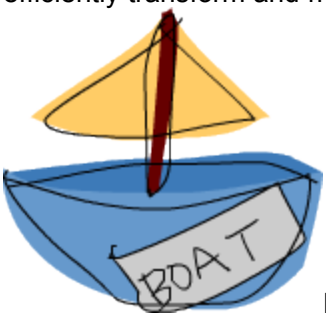
So what's Haskell?

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.

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. For the purposes of this tutorial we'll be using GHC, the most widely used Haskell compiler. The best way to get started is to download the Haskell Platform, which is basically Haskell with batteries included.
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.

Starting Out

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
```

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.

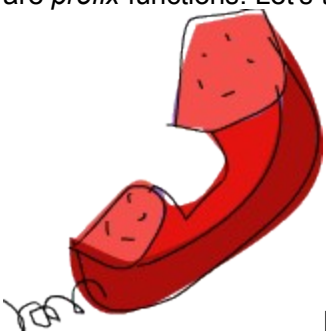
```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
      True
```

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!). `min` returns the one that's lesser and `max` returns the one that's greater. See for yourself:

```
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))`.

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.

```
1. 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 saved 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.

```
1. ghci> :l baby
2. [1 of 1] Compiling Main             ( baby.hs, interpreted )
3. Ok, modules loaded: Main.
4. ghci> doubleMe 9
5. 18
6. ghci> doubleMe 8.3
7. 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.

```
1. 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).

```
1. ghci> doubleUs 4 9
2. 26
3. ghci> doubleUs 2.3 34.2
4. 73.0
5. ghci> doubleUs 28 88 + doubleMe 123
6. 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:

```
1. 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!

```
1. doubleSmallNumber x = if x > 100
2.                         then x
3.                         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.

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

Had we omitted 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.

```
1. 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.

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

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.

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

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

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

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

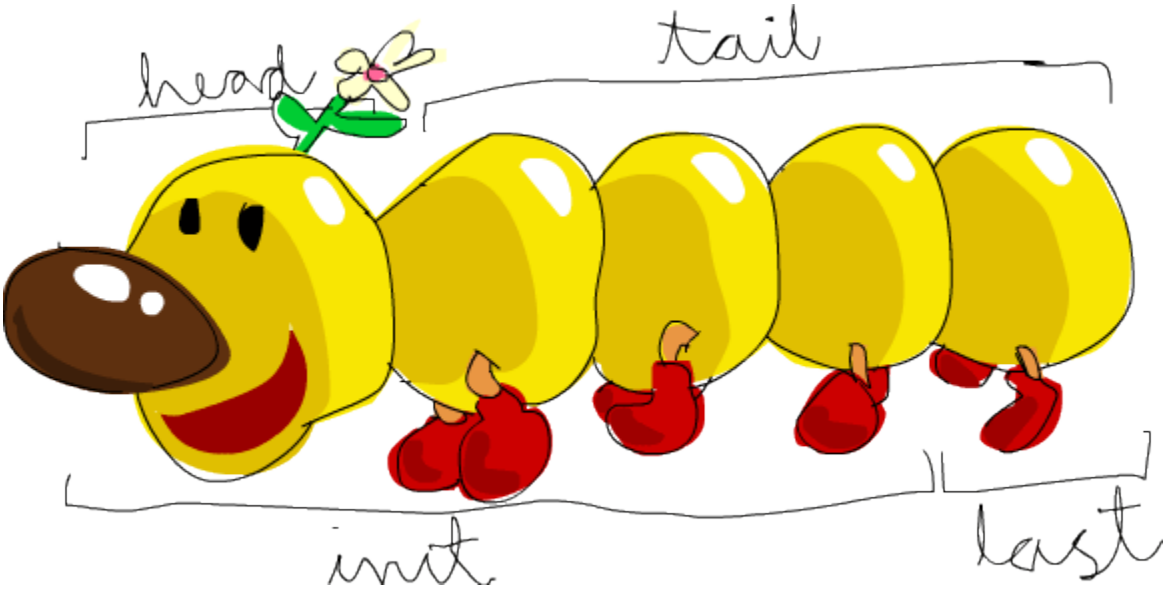
`last` takes a list and returns its last element.

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

`init` takes a list and returns everything except its last element.

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

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



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

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

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

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

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

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

`reverse` reverses a list.

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

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

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

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

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

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

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

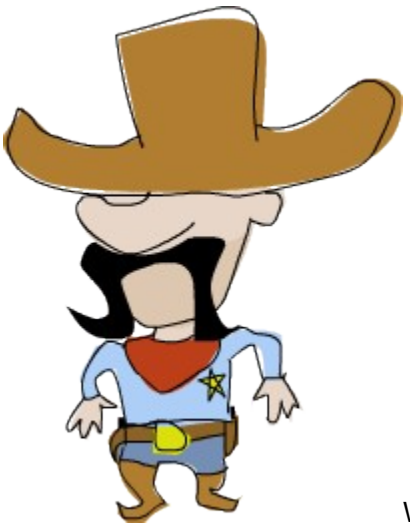
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

`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 later

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> [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]`**.

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 $S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$. 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 <- n
ouns]
["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 comprehensions 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.

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 problem 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:

```
1. Couldn't match expected type `(t, t1)'
2. against inferred type `(t2, t3, t4)'
3. In the expression: (8, 11, 5)
4. In the expression: [(1, 2), (8, 11, 5), (4, 5)]
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 represent a wide variety of data. For instance, if we wanted to represent someone's name and age in Haskell, we could use a triple: `("Christopher", "Walken", 55)`. 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 tuple 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.

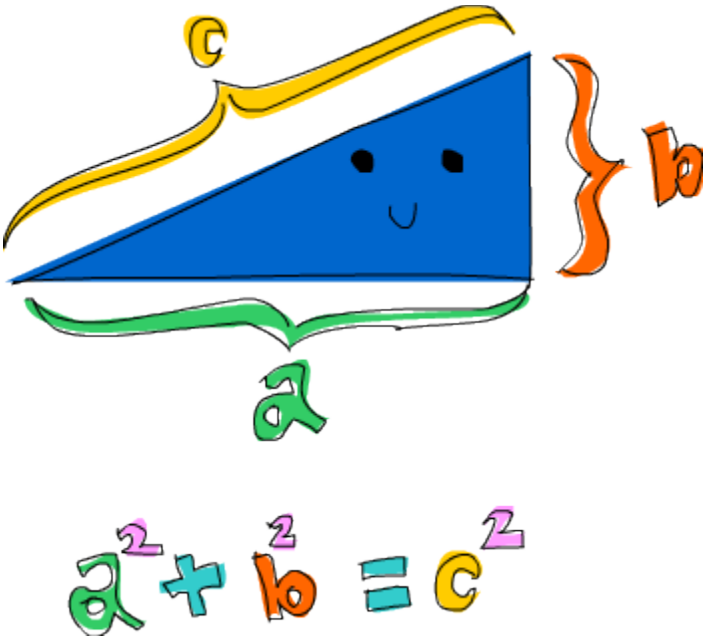
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

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:

```
1. 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.

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

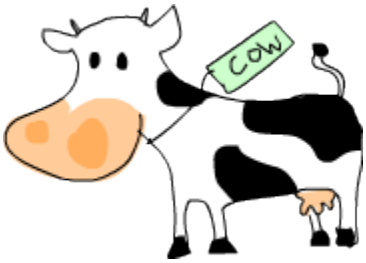
We're almost done. Now, we just modify the function by saying that we want the ones where the perimeter is 24.

```
1. 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]
2. ghci> rightTriangles'
3. [(6,8,10)]
```

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

Types and Typeclasses

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.

```
1. removeNonUppercase :: [Char] -> [Char]
2. 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:

```
1. addThree :: Int -> Int -> Int -> Int
2. 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.

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

1. ghci> factorial 50
2. 30414093201713378043612608166064768844377641568960512000000000000
```

`Float` is a real floating point with single precision.

```
1. circumference :: Float -> Float
2. circumference r = 2 * pi * r

1. ghci> circumference 4.0
2. 25.132742
```

`Double` is a real floating point with double the precision!

```
1. circumference' :: Double -> Double
2. circumference' r = 2 * pi * r

1. ghci> circumference' 4.0
2. 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: **`()`**

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!

```
1. ghci> :t head
2. 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.

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:

`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 it 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.

Syntax in Functions

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.

```
1. lucky :: (Integral a) => a -> String
2. lucky 7 = "LUCKY NUMBER SEVEN!"
3. 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:

```
1. sayMe :: (Integral a) => a -> String
2. sayMe 1 = "One!"
3. sayMe 2 = "Two!"
4. sayMe 3 = "Three!"
5. sayMe 4 = "Four!"
6. sayMe 5 = "Five!"
7. 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 nutshell, this is what happens if we try to get 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 fu
nction 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) and 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 occurred. 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)convenient 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 head. 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
```

```
3. 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:

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

1. ghci> capital "Dracula"
2. "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.

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)

```
1. bmiTell :: (RealFloat a) => a -> String
2. bmiTell bmi
```

```
3.      | bmi <= 18.5 = "You're underweight, you emo, you!"
4.      | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're u
   gly!"
5.      | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
6.      | 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 conditions. 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.

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

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

```
1. ghci> bmiTell 85 1.90
2. "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

ghci> 3 `myCompare` 2
GT
```

Note: Not only can we call functions as infix with backticks, we can also define them using backticks. Sometimes it's easier to read that way.

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, yo
u!"
    | 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, f
atty!"
    | 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 u
gly!"
    | 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 parameters (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.

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 be 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 to some that's more readable.

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] -> 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 l
ist."
                                               xs -> "a longer list.
"
```

They 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."
```

Recursion

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.

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.

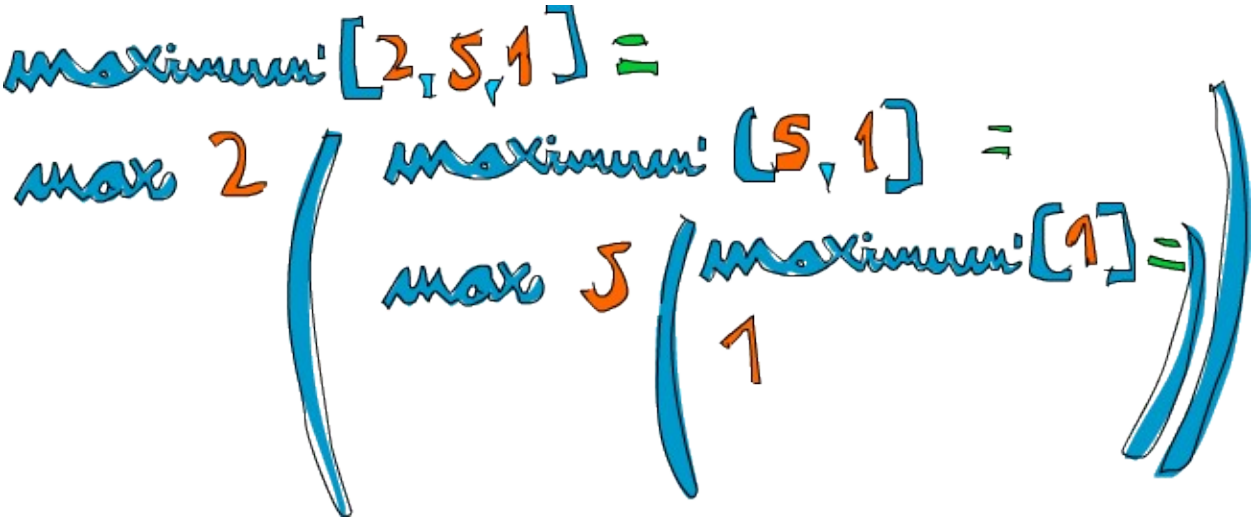
```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

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.



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.

```
1. reverse' :: [a] -> [a]
2. reverse' [] = []
3. 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.

```
1. repeat' :: a -> [a]
2. 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`.

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

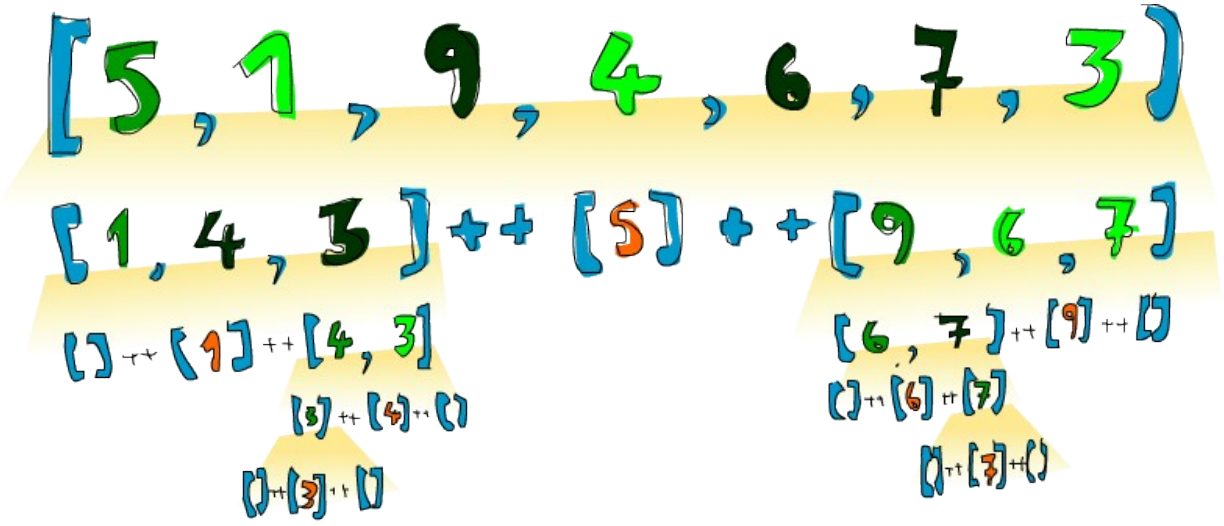
comes the main algorithm: **a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).** Notice 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:

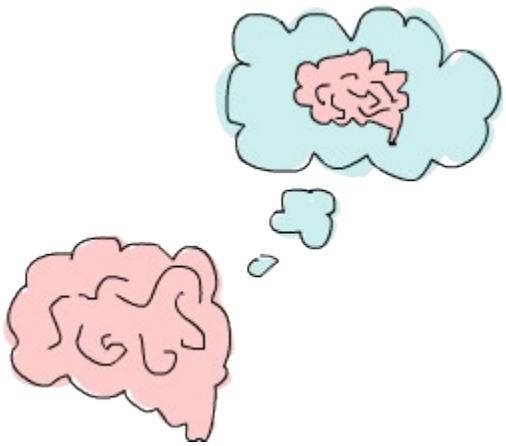


An element that is in place and won't move anymore is represented in **orange**. If you read them from left to right, you'll see the sorted list. Although we chose to compare all the elements to the heads, we could have used any element to compare against. In quicksort, an element that you compare against 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.

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.

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.

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 **function application**. 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:

```
1. divideByTen :: (Floating a) => a -> a
2. 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:

```
1. isUpperAlphanum :: Char -> Bool
2. 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?

```
1. ghci> multThree 3 4
2. <interactive>:1:0:
3.     No instance for (Show (t -> t))
4.       arising from a use of `print' at <interactive>:1:0-12
5.     Possible fix: add an instance declaration for (Show (t -> t))
6.     In the expression: print it
7.     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.

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!

```
1. applyTwice :: (a -> a) -> a -> a
2. 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' _ [] _ = []
```

```
3. zipWith' _ _ [] = []
4. 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:

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

```
1. flip' :: (a -> b -> c) -> (b -> a -> c)
2. flip' f = g
3.     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]
```

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 comprehension. `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 diFfeRe
nt"
     "uagameasadifeent"
     ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r a
LL 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.

```
1. largestDivisible :: (Integral a) => a
2. largestDivisible = head (filter p [100000,99999..])
3.     where p x = x `mod` 3829 == 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:

```
1. ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
2. 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) [n^2 | n <- [1..], odd (n^2)])
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> (listOfFuns !! 4) 5
20
```

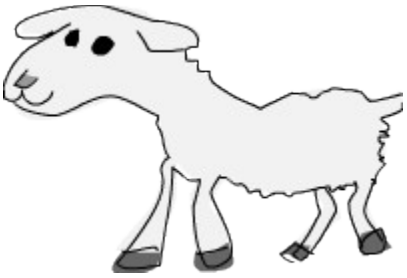
Getting the element with the index `4` from our list returns a function that's equivalent to `(4*)`. And then, we just apply `5` to that function. So that's like writing `(4*) 5` or just `4 * 5`.

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
```

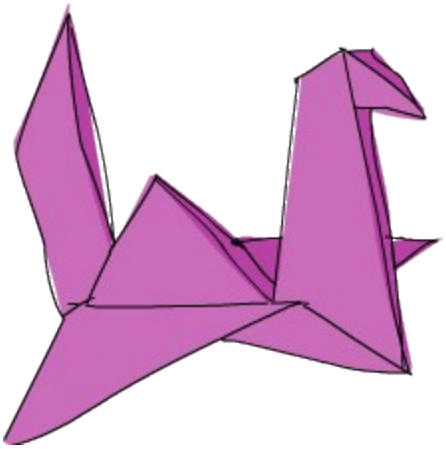
If we define a function like this, it's obvious why the type declaration is what it is. There are three `->`'s in both the type declaration and the equation. But of course, the first way to write functions is far more readable, the second one is pretty much a gimmick to illustrate currying.
However, 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.

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.

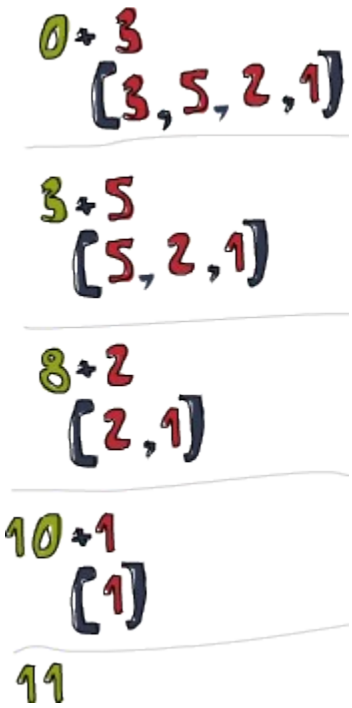
First let's take a look at the `foldl` function, also called the left fold. It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next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

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

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

Testing, one two three:

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

```
1. sum' :: (Num a) => [a] -> a
2. 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.

```
1. elem' :: (Eq a) => a -> [a] -> Bool
2. 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.

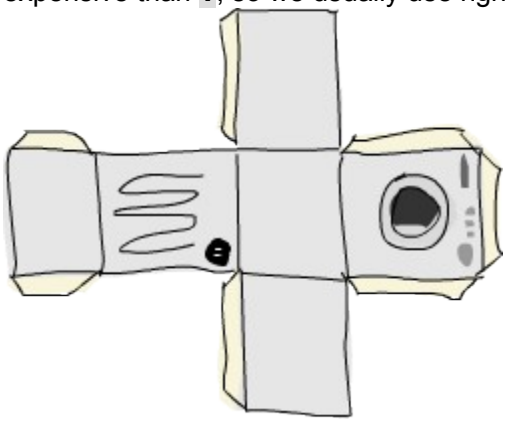
The right fold, `foldr` works in a similar way to the left fold, only the accumulator eats up the values from the right. Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so `\acc x -> ...`), the right fold's binary function has 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.

```
1. map' :: (a -> b) -> [a] -> [b]
2. 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 it 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` is better implemented by pattern matching, but this just goes to show, you can still achieve it 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.

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]
```

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.