Happy Learn Haskell Tutorial Vol 1

http://www.happylearnhaskelltutorial.com/
Written and Illustrated by *GetContented*
*(*enquiries@getcontented.com.au*)*

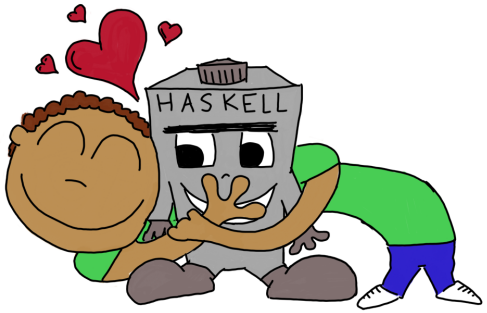
Updated on: June 5, 2016.

Contents

1 How to learn Haskell enjoyably

1.1 Fascination

Just like you, when we discovered computers we were taken in, fascinated by their potential. We watched happily as programs seemed to make anything possible. Such amazing works by the programmers who created them.



1.2 Wish to Create

Delight quickly changed to desire: desire to write our own programs, but how best to begin? Discovering many books, we filled our heads with knowledge, sadly not finding much guidance about proceeding with the practical craft.

1.3 Too Many Details?

Trying our hand at making programs ourselves, we discovered details cluttered us. Losing our delight, we'd stop. We simply didn't understand how to use our dusty book-learning. Our programs stank.

1.4 The Journey Begins

We felt bound by our knowledge, needing to return to the freedom of before we began, so we endured tedious practice and failure, making the theory our own, slowly understanding. Again we found no guide lighting our path, but after an arduous journey, finally the fog

lifted and we could write excellent programs and still be excited and joyous.

1.5 And Now You...

So you too want to become a programmer. Luckily, you've found this guide, crafted by people who have lived this path to the end, and would save you the pain and boredom we endured.

1.6 No Magic, But Why Pain?

There's no magic here, but there are more-, or less-difficult paths to choose. If you want mastery, you always need deep practice, but why should this mean pain and boredom? Small steps will be our guide, and fun our companion. But, how to journey?

1.7 Precise Language

Far more precise language than ours is needed to program. To write in such a language, we need to know the correct words and their strict arrangements; and how to tease our intent apart and clothe it nicely as a program. This is not all.

1.8 Taking Care

A little knowledge, that dangerous thing, produces some success, and worlds of possibilities arise, bringing excitement with them. The eager beginner quickly gets into a flurried muddle, as enthusiasm has them tackling too much too soon. Initial elation slowly turns into bitter disappointment and they give up or worse, spread hate. We don't want this for you.

1.9 Two Phases of Learning

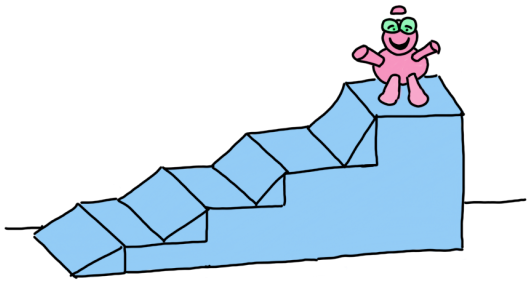
Instead, your learning will proceed in two staggered stages. Each lesson will introduce several programs. Reading, understanding and typing these in yourself will embed them in your own experience. You need this practice to recognise the pieces and to know what they do. We'll then adjust them slightly and see how they change.

1.10 Stages Build Skill

This proceeds in a graded, staged way. Using real-world problems to illustrate simple solutions with the language constructs seen so far, our reading will slowly increase in difficulty until we have seen the core of the language and beyond.

1.11 Simple, but Fun Examples

The second stage starts further along, when enough reading means you know how to do small things. Again, we'll take care to work within enjoyable limits as we show you how begin to build a path the other way: from problem solving, to intention, to code. This stage will solidify your understanding. Proceeding, we slowly take the training wheels off and before you know it, you'll be able to make some well-designed programs that read well, are easy to understand and are enjoyable to change.



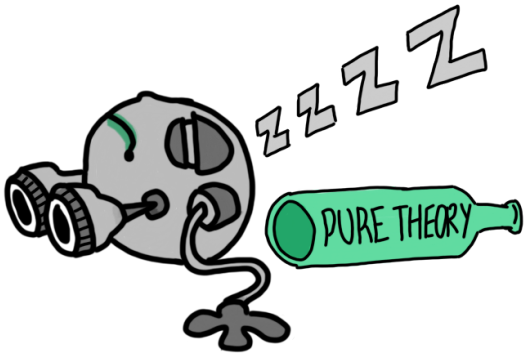
Many books don't address the subject of how to craft solutions, or they just leave you with nothing but some exercises and your own intuition. Most are focussed heavily on programming language topics first, and how you use them to do things second, if at all. You'll notice we're primarily interested in you learning how to do useful things, rather than the language for its own sake. Practical things will anchor the language more in your memory and experience.

1.12 Progressive Learning

Our material is cleverly crafted to gradually introduce you to the entire language. We do this over the course of the various, interesting examples which are present in every chapter, across all the volumes. We chose this way because the other way bores people to sleep, which is inconsiderate and tedious.

1.13 Natural Assmiliation

Countless people have found programming difficult to learn because of boring examples, unpolished writing, or material being organised for language features rather than learner interest.



On the other hand, we've seen great success in material that uses varied repetition, amusing useful pictures, fun examples using real-

world topics and small graded steps. This guided our choices in building the entire series, one volume of which you have in your hands.

1.14 Motivation is King

If motivation really is what pulls us through practice, then we sincerely hope we've inspired motivation and excitement in you with this series, and wish you never forget the love that we all share for learning and programming. May it guide you always.

2 Your First Step

Let's begin our journey to learn Haskell with a program that gets a message on the screen.

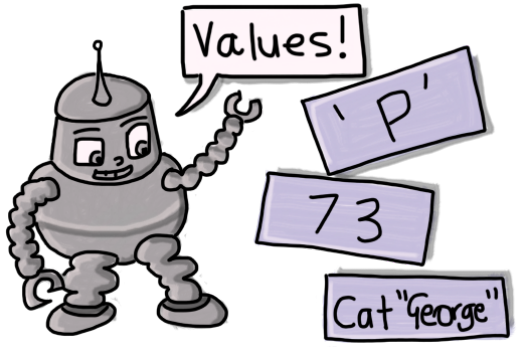
It's important to begin simply, because doing small steps helps you to avoid frustration and increases happiness!

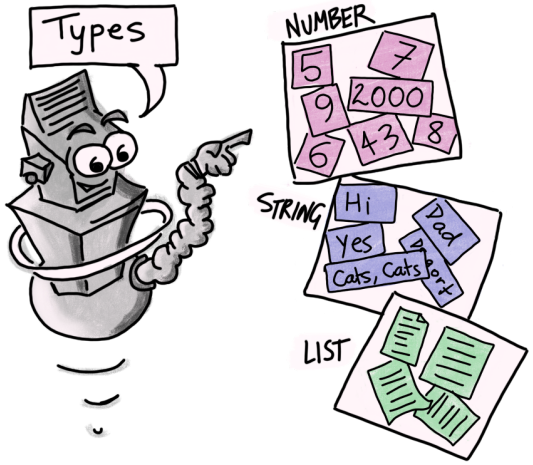
So, in this lesson, we'll discover this very simple program together, and pull it apart to understand it. None of the following sentence will make any sense yet, but it will by the end of the lesson: our first program will be a single **definition** for an entry-point called `main`, and it will use a built-in **function** called `putStrLn` and a **value** of **type** `String`.

So, we'll first explain what the terms **values**, **types**, **definitions** and **functions** mean when talking about Haskell, then we'll show you the program, and explain it piece by piece.

2.1 Values

A **value** in Haskell is any data. The word `"Step"`, for example. All values have a **type**, which tells us and Haskell what sort of data they are. (The type of `"Step"` is `String`).





2.2 Types

A **type** is a name for a set or group of values that are similar. As an example, there is a type for representing values like `5`, and `34` called `Integer`. Most of the time, the types of our values are not written down in the program. Haskell will work out what the types are for you, however, Haskell lets you write them down if you'd like to, like this: `5 :: Integer` and `34 :: Integer`.

We'll definitely learn much more about types later, so don't worry too much about them for now. To whet your appetite, though, Haskell lets us make our own types. This means it'd be impossible to list them all here, but you can easily write most programs using only a handful of types, so it's not too hard to think about.

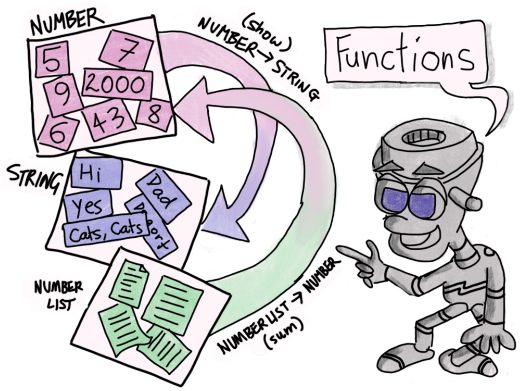
Types are important because they decide how the pieces of your program can fit together, and which values are allowed to go where.

2.3 Definitions

The "`=`" symbol lets us tell Haskell that we want it to remember a name for an **expression**. The name goes on the left of it, and the expression on the right. An expression is simply a way to connect

up some values together in relation to each other.

The = symbol relates (or we can say **binds**) the name or pattern on the left of it with the expression on the right of it. For example, `five = 5` tells Haskell that the name `five` means the value `5`.



2.4 Functions

A **function** is a relation between one type and another type, and they're used in expressions with values. In Haskell a function is itself a value. If you provide a function with an input value, (by **applying** the function to the value), it will **return** (give you back) the corresponding output value of the output type when it's evaluated.

Be careful how you think about this! Function means something different in Haskell than in most other programming languages. Functions in Haskell are not sets of steps for the computer to follow.

Our first program will show you an example of function application, so read on!

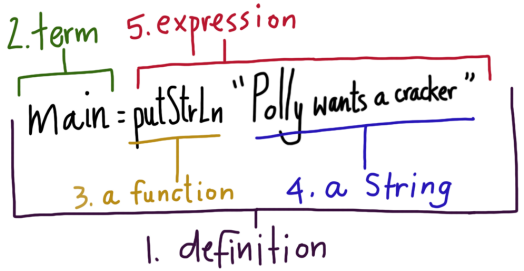
2.5 Our First Program

Here, then is our first program to read. Just one line. We see an **expression** that is being named "**main**". The expression involves a **function** being **applied** to a **value**, and the **types** are not shown:

```
main = putStrLn "Polly wants a cracker"
```

If you were to run this first program, it will put `"Polly wants a cracker"` on to the screen.

This may look very confusing, but keep calm. We'll break it into five parts, and explanations:



2.5.1 A Definition

This whole line is called a definition. It's made of three pieces: 1. the variable or term name, 2. the "=" symbol, and 3. an expression.

Haskell programs are created by making definitions and expressions and embedding expressions in other definitions and expressions.

It's important to realise that the "=" symbol doesn't mean we're "setting" anything here, or "putting" anything into anything else. We use it to name expressions, is all. The name does not "contain" the value, so we cannot re-define the same name at a different point in our program, or change the name's "content" or definition after it has been defined.

2.5.2 A Term, or Variable Name

main: This piece is the name (called the **term** or **variable**) that we're defining. In this particular case, it's "`main`", and that is a special name to Haskell. Every program must define `main`, and it is evaluated and executed by Haskell to run your program. If you named it `pain` instead of `main`, your program would not work. This is called the **entry point** because it's where Haskell will start executing your program from.

main is a value, and also an input/output action (IO action for short). IO actions are different to normal values because they describe how to make input & output happen. We'll learn more about this later.

2.5.3 A Function Name

`putStrLn`: this is a function name. If we put the name of a function in an expression and a value to its right like this, when it's evaluated, Haskell will apply the function to the value to "produce" another value. We say it **takes** a String value, and **returns** an IO action. When this IO action value is **executed**, it will output its String to the screen.

2.5.4 A String Value

`"Polly wants a cracker"`: This is one way to write text in Haskell programs. It is a value whose type is `String`. (We can just

call this a `String`). A `String` is a List of `Char` values. `Char` values, or characters, are any written letter, number, or symbol. If you like, you can imagine a `String` as symbols pegged to a piece of yarn — strung together. We're using this `String` in our program to provide the `putStrLn` function with the data it needs to print to the screen.

2.5.5 An Expression

Expressions are how we **express values**. They have a type, too. This particular expression evaluates to the IO action that results from providing the `String` `"Polly wants a cracker"` to the `putStrLn` function.

This is a lot to take in all at once. Study it well, but don't worry if things aren't clear yet, it will become easier as you read on. Check back to this page later for reference.

2.6 Homework

Your homework is to try to run this program, if you have a computer. This will involve setting up Haskell, too! Look at Chapter 21 if you want some helpful hints on how to get started and set up your Haskell development system.

3 Types as Jigsaw Pieces

Ok so we previously looked at reading our first program and began to understand it. Let's look at some more programs and expressions. This time, though, we'll focus on the **types** of the elements to explain what's going on.

3.1 A String Value

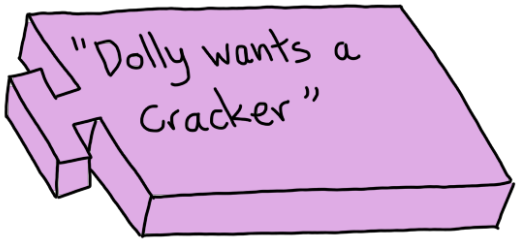
Here is a `String` value:

```
"Dolly wants a cracker"
```

A nice way to think about values is as if they're magical puzzle pieces that can shrink and grow as needed to fit together. If values are puzzle pieces in this analogy, then their **types** would be the shapes of the puzzle pieces. This analogy only really works for simple types, but it can be handy.

3.2 Puzzles

So, perhaps we might think of that `String` value like this:

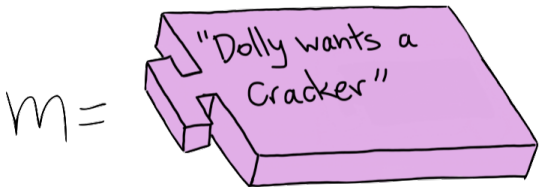


3.3 Definitions again

Let's look at a definition for this `String` value. We make definitions so we can re-use things in other parts of our program later on. We'll name this one `m` (as in message):

```
m = "Dolly wants a cracker"
```

As we've seen briefly before, this is telling Haskell that we want to **define** the name `m` as being the `String` that is literally `"Dolly wants a cracker"`. Writing definitions for names like this is just like writing a mini dictionary for Haskell, so it can find out what we mean when we use these names in expressions in other parts of our programs.



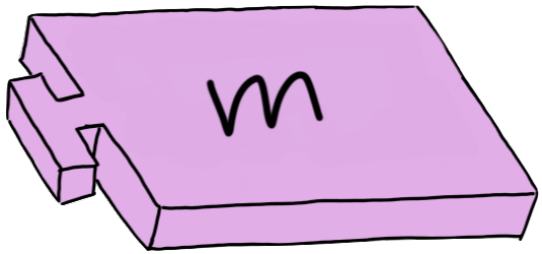
3.4 Type Annotations

Now we'll see what it looks like when we include the code to describe the **type** of `m`, too.

```
m :: String
m = "Dolly wants a cracker"
```

Writing the **type annotation** like this helps both us and Haskell to know what **types** values and expressions are. This is also called a **type signature**. You probably worked out that `(::)` specifies the type of a name in the same way that `(=)` is used to specify the meaning of a name.

Haskell now knows that `m` means `"Dolly wants a cracker"` whenever we use it in another expression, and that it is a `String`. Anywhere we use the `m` variable, Haskell will use our `String` value; they now mean the same thing.



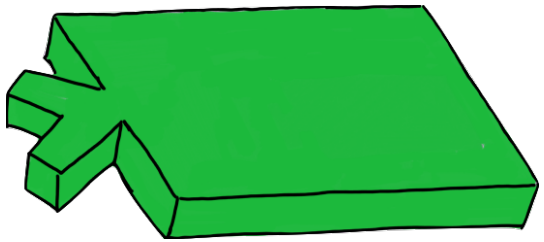
3.5 Types for Functions

Now, what about `putStrLn`? As we know, this is a name that Haskell already has ready-made for us. Let's look at its type:

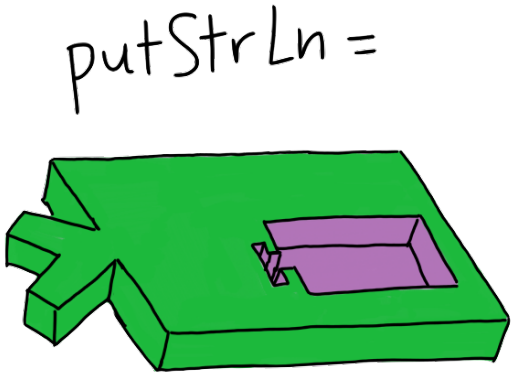
```
putStrLn :: String -> IO ()
```

(**Note:** You will never need to write this type in your own code because it's already defined. We've just shown it here so you can see what it is, and how to work with it.)

We can read this as "`putStrLn` has type `String` to `IO` action", or "`putStrLn` is a function from `String` value to `IO` action". `String` is the function's input type, `(->)` signifies that it's a function, and goes between the input and output types, and `IO ()` is the function's output type.



What about `putStrLn`? Well, it'd need be a value of type `IO ()` once it had a `String` popped into it. We could imagine that it looked like this, roughly:



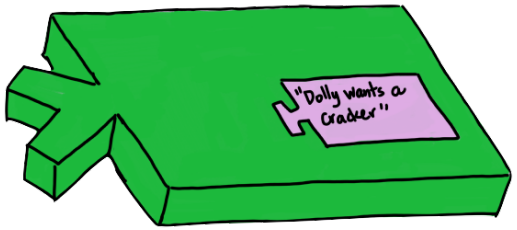
3.6 IO Actions as Puzzles

Let's look at what an `IO ()` value might look like if we imagined it as a puzzle piece (the shape is completely made up, but we'll use it to explain the types of values):

It's not an `IO ()` value. It's something that can be, when supplied with a `String` value. See how that makes it a kind of mapping between values of these two types?

3.7 Putting values together like puzzle pieces

When we put a `String` value in that pink gap, we get an expression that will evaluate to a value whose type is `IO ()`.



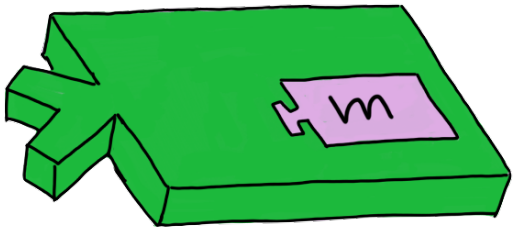

```
putStrLn "Dolly wants a cracker" :: IO ()
```

As you can see, we can put a type signature on almost any expression. For example, here's another expression that means the same thing, but with too much annotation:

```
putStrLn ("Dolly wants a cracker" :: String) :: IO ()
```

We put a sub-expression type annotation for the `String` as well as an annotation for the whole expression! Very silly. In real code, we'd never see an expression like this, because Haskell can almost always work out the types from the context, which is called **type inference**. Notice we're using parentheses around the string so Haskell knows which signature goes with which expression.

Of course we could use our `m` that we defined earlier instead, and show off its type annotation, too:




```
putStrLn m :: IO ()
```

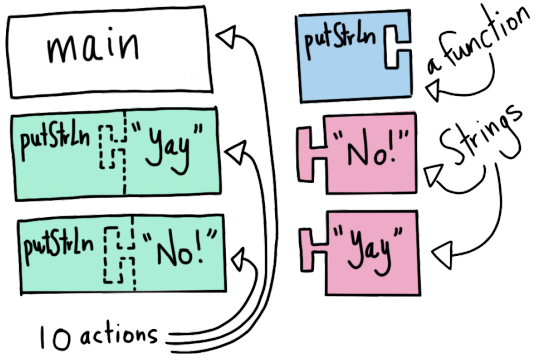
3.8 The whole program

So this expression is an `IO ()` value, and because `main` needs to be an `IO ()` as well, we can see that one possible definition of `main` could be this `putStrLn` expression. Here's the whole program to print out the `String`:

```
m :: String
m = "Dolly wants a cracker"

main :: IO ()
main = putStrLn m
```

3.9 Another way to look at it

So far so good, now let's see some similar things visualised slightly differently:



When we put a value next to a function, like in the expression `putStrLn "No!"`, we can say `putStrLn` is taking the literal `String` whose value is `"No!"` as its **argument**. The word argument means the parameter to a function. In Haskell, wrting two things with a space between usually means that the thing on the left is a function and it will be applied to the value or expression on the right.

3.10 What is a Function?

A function is a value that expresses a relationship between types. That is, it relates one set of values to another set of values. The `putStrLn` function relates `String` values to `IO` actions, for example. If you plug (or connect) a `String` value into `putStrLn` as we have seen above, together they form an expression of type `IO` action.

3.11 Arguments?

The "hole" position that `putStrLn` has is called an **argument** (or parameter). By giving a value to a function as an argument as described, we are telling Haskell to connect them into one expression that is able to be evaluated, into a value of its **return type**.

In the picture above, `main` is shown as being square-shaped. By itself, `putStrLn` doesn't have the same shape as `main`, so it needs something "plugged" into it before we can equate it to `main`.

Once the `String` is plugged in, the whole thing is an expression with the same shape that is required for `main`, which means we can write a program with the definition for `main` we saw above.

```
main :: IO ()
main = putStrLn "No!"
```

Another way to say this is that the `IO` action that results from

connecting these together is a value. Before it has the `String` connected to it, it's an `IO` value that is a function of its `String` argument (and that function is named `putStrLn`).

3.12 Nonsense Programs?

It's important to remember that Haskell receives programs as text files, so there is nothing stopping you from writing programs that make no sense, such as trying to provide something other than a `String` as an argument to `putStrLn`.

```
main :: IO ()
main = putStrLn 573 -- this will not compile!
```

Haskell won't let you compile or run obviously incorrect programs, though. It will give you an error if you try. You should try to compile the incorrect program above. Haskell will tell you there is a type-checking error.

3.13 The shape of main

So, we saw that `main` needs its definition to be of a certain "shape". Haskell requires that `main` is an `IO` action. Its type is written `IO ()`, which is an `IO` action that returns nothing of interest (but does some action when executed). We use the double-colon operator `(::)` to mark the type that a definition, expression or value is "of". Below, we'll see this in operation some more.

3.14 The two simple programs

Let's take a look at these two programs which were referenced in the drawings earlier:

```
main :: IO ()
main = putStrLn "Yay"

main :: IO ()
main = putStrLn "No!"
```

By the way, you can only define an identifier **once** in each Haskell file, so don't try to put both the above definitions in one file and expect it to compile. Haskell will give you an error. An identifier is just another name for a variable or term, as discussed in an earlier chapter.

The first program prints out "Yay" on the screen, and the second one prints "No!"

As discussed we read the `(::)` operator as "has type". `IO ()` is the type of `IO` actions that wrap the empty tuple. The empty tuple is a container that can never have anything inside of it, so it's used as a simple way to express the value of having no value at all.

Handily for us, `IO ()` is the same type of value that `putStrLn` returns when we give it a `String` as an argument.

3.15 Pulling the definitions apart more

Let's read another program that does exactly the same thing as the No! program above, but goes about it a little differently:

```
message :: String
message = "No!"

main :: IO ()
main = putStrLn message
```

We've just pulled the No! String out into its own definition (naming it `message`), with its own type signature. Don't let it scare you, it's pretty simple. It just means `message` is a `String`, and its value is `"No!"`.

3.16 Are signatures mandatory?

Do you have to write type signatures? Not always! We started the book with a definition for `main` that had no type signature, and then added one, so you can leave them off — as long as Haskell can infer what you mean by itself. Don't worry, it'll tell you if it can't work it out.

Haskell uses a pretty nifty feature called **type inference** to figure out what types things should be if you don't write type signatures for them.

3.17 Signatures as documentation

However, it's often a good idea to put type signatures in so you and others know what your code means later. We recommend doing this because it improves the readability of your program, too.

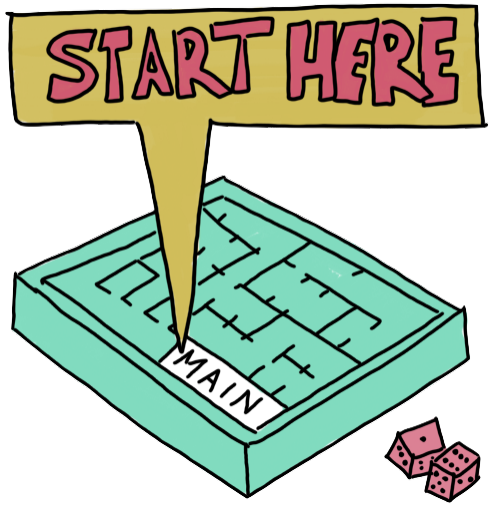
Did you notice you can use type signatures to let you work out what to plug in to what? They can be incredibly useful when programming.

3.18 Homework

Your homework is to do an internet search on Haskell programs and see if you can identify at least ten definitions, and ten type signatures. Don't get too worried by odd looking things, just stick to the homework. We want to get you familiar with what these things look like.

4 The Main Road

We discovered that `main` is the entry-point of all Haskell programs. In other words, the point where they start executing. We also discovered that `main` is an action, so now we're going to delve a little bit into actions, `IO`, and what purity means.



Bear with us while we work through this part. Haskell works differently than you would think, and it's important to get a clear mental picture of how it behaves.

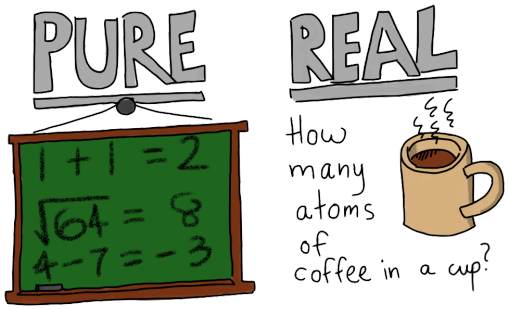
4.1 How a Haskell Program is Made

Haskell programs are made by writing definitions in text files, and running a program called a **compiler** on those text files. The compiler understands Haskell, and translates the text files as Haskell into an executable program, which we can then run.

4.2 Purity

In Haskell, all the things you can write are called **pure**: definitions, values, expressions and functions, even the values that produce actions.

Pure here means they are consistent, equational and express truths about value. That makes them easy to think and reason about. They're expressed in the world of the computer, where pure thought can exist. All the expressions in Haskell use this purity, which makes writing software joyful because we can lock parts of our programs down into separated, predictable behaviour that almost always works as we think it will.



To understand this, let's think about the way addition works. It never, ever changes. If you add the numbers 1 and 5, you will **never** get a different result than 6. Of course! This is what we mean by **pure** — we can talk about what is true or false in this "world" with some certainty. Contrast this to something from the real world: the current time. This is **not** a pure value because it is **always** changing.

Let's see a program that does this. Don't worry if you don't understand anything about it yet. We'll explain it in later chapters:

```
-- a program to
-- add 1 and 5

main :: IO ()
main = print (1 + 5)
```

Every time you run the program above, it will return 6. It can **never** change, which is because it only uses simple pure functions and pure values to obtain its result. Contrast that to the following program, which uses pure **IO actions** that **contain** non-pure values:

```
-- a program to get and print
-- the current time as seconds

import Data.Time.Clock.POSIX

main :: IO ()
main = getPOSIXTime >>= print
```

When we run **this** program, it takes the current time from `IO`, and passes it to some screen-printing code which is built into the `IO` portion of the `print` function, which prints it out on the screen. Every time you run this program you'll get a different answer, because the time is not a **pure** value. It's always changing.

Something to note, though, is that `getPOSIXTime` and `print` are still pure functions. They will always return the same thing: a **description** of non-pure actions. That is, they **describe** non-pure values and functionalities. That means when we write our program we can still use pure functions like the above, but when it is run, those values can send non-pure values and functions around, like the time or printing things to the screen. These are called **IO actions**, because they describe some **action** on the input/output context.

4.3 Everything in Haskell is pure

Some more about this. So how can we say everything in Haskell is pure, if we just showed you a Haskell program that deals with the time which we just explained is an impure value? Well, the Haskell **program** is pure, and only expresses pure values and functions, but the IO type allows us to describe and contain computations and values that deal with the non-pure real world!

Pure values and expressions and functions are much easier to reason about because they're simple and direct, which is why it's nice to program in Haskell: It lets us talk about the non-pure world using pure descriptions.

Unfortunately, most of the interesting things we want to program are not completely pure. Haskell has certain special functions and values that are marked as `IO` actions. You've seen one or two already. These are also pure, like everything in Haskell, but they give us a gateway into the messier "real world". They let us describe **actions** that can happen in the real world.

In Haskell, the types of things that let us do this are made with a constructor named `IO`, which stands for Input/Output. That is, the world of the error-prone, non-pure, complicated, real world where things are always changing, and where there is input and output from keyboards, mice, disks, time, the internet, random numbers and printers. The real world is usually hard to think about simply, and much more difficult to program for. It doesn't obey the same rules as the pure world and is harder to lock down into predictable behaviour because it's so complex and complicated.

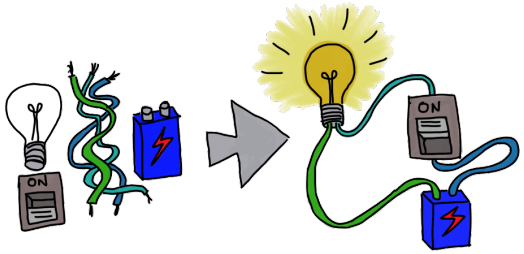
```
getPOSIXTime :: IO POSIXTime
```

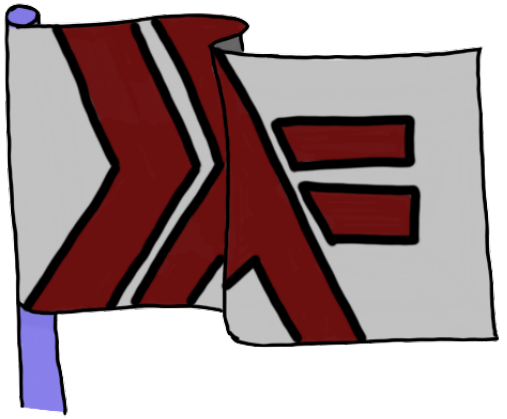
As we can see, the type of the function that can get the POSIX time (a type of time value) is an IO type, which shows that it is an action in the impure world. That means it will return a POSIXTime when it is **executed**.

So, `IO` actions can be **executed**, which is what happens when their `IO` behaviour is activated - this is how programs are run. This `IO` behaviour is effectively invisible to the pure world. The pure world can never "see outside" to the real `IO` world, however the `IO` world can use pure values, functions and expressions without problem. Running a program is how we execute `IO` actions. When we compose our `IO` actions with other `IO` actions and pure functions, we can build up useful, functioning programs.

4.4 An Analogy

Let's use an analogy here: think about a simple electric circuit: a battery, some wires, a switch and a light globe. By themselves, all of these things are simple, "pure" things. The light and switch are slightly different than the others, but they are still just themselves. However, the switch can "do input" into the circuit, and the light globe can "do output" from the circuit in the form of glowing light. The wire and battery can do no such things. They don't have any **action** in the real world at all.





If we connect these components into a circuit by binding them together, we can throw the switch (that is, give the circuit some input) and the light bulb will cause there to be light (that is, we'll get some output). This doesn't change what these components are, obviously. However, some of the parts provide effects in the world outside the "world" of their pure value inside the circuit. The wires are like our totally pure non-`IO` functions in Haskell because they don't do anything outside the circuit: they're not like the light or the switch.

You can see how **all** of these components are in themselves purely what they are, and **all** of them can be used as part of other bigger composed circuits, which in total have some `IO` action on the real world, but also that **some** of these component (such as the wires) don't actually do any actions in the world of `IO` for the circuit, but are necessary for the whole circuit.

In a similar way, `IO` actions have a foot in both worlds: while being able to be evaluated like ordinary Haskell expressions, which does nothing in the `IO` world, they also contain the ability to be executed: that is, to interact with the outside world and effect it in some way, and so they are marked with the `IO` type marker. To create programs that interact with the `IO` world, we bind combinations of these `IO` actions together along with pure functions to create bigger `IO` actions that can do what we want.

4.5 Haskell is Awesome

Haskell is an awesome language: it's capable of letting us cleanly think about and build pure expressions and functions, and also lets us connect these pure things up to the real, messy world in a way

that keeps as much of our programs simple, clear and separate as possible.

When a program is composed of pieces like this, it's very easy to reuse parts of it, and it's much easier to spot problems and adjust things.

Note that Haskell programmers will often refer to a pure `IO` value as an action, as well as referring to the part of one that effects the real world, such as the action of putting a message on the screen. This can sometimes be confusing, so just remember it can mean either.

5 Function Magic

One of the simplest types possible has only two values. They represent truth and falsehood. In Haskell, this type is called `Bool` (short for boolean), and its two values are named `True` and `False`:

```
True :: Bool
False :: Bool
```

In later chapters, we'll see how important these two values are. We'll use them for checking things, and doing all sorts of other useful things. For now, we're going to use them to introduce you to how to really think about **functions**.

5.1 A Story of Magic Coins

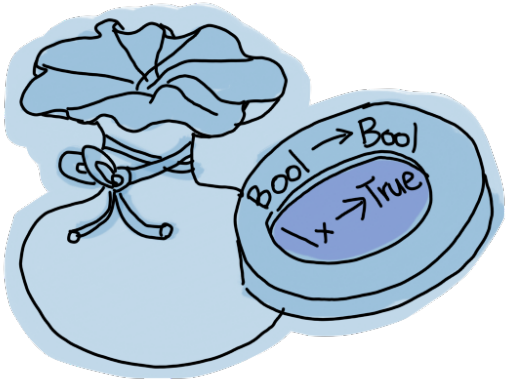
Let's imagine you're a book-keeper on holidays looking for a good spot to read a pocket book called "The magic language of Haskell", and you end up in a milk bar on planet Boolean where the local currency is `Bool` coins. You just bought a milkshake, and now you really want to play your favourite song on the jukebox as you read, so you open up your money holder, and here's what you see:

Drats! Mostly `False` coins. The jukebox takes two `True` coins, and you only have one of them.

You spy a misshapen robot character at one end of the bar who might be a Haskell wizard. Maybe this guy can help.

He puts down his milkshake, and as you explain your plight to him, he stops frowning and clicking his tongue and reaches into his cloak. Pushing a small glowing bag in your direction, he seems genuinely happy to be able to help.

"You pay for these by learning how they work" he says simply, refusing your offer of paying with False coins. Satisfied, you take the bag and look inside as you walk to the jukebox. You pick out one of the glowing coins inside, and take a look. It appears to be made of something magical:



You notice an inscription on the coin. It looks like some kind of arcane magical writing, obviously written in Haskell, the ancient magical language of the robot wizard folk:

```
(\x -> True) :: Bool -> Bool
```

You're a little familiar with this language yourself. Being a bookkeeper, you like to read many of the titles you stock, some of them

about Haskell. Happily you remember you just happen to have that beginner's pocket guide to the Haskell language in your back pocket. Oh yes, that was why you came in here in the first place, to read it!

You recall that the arrow symbol `(->)` usually indicates a function: that is, a mapping from one **type** of things to another **type**. However, here there is a `True` **value** on the right, and `\x` on the left. These aren't types. What's going on?

You consult your pocket book, and after a few minutes of rifling through the pages, you discover that this is a thing called a **lambda**, in coin form. A lambda, you read, is a function with no name: a way of getting a completely new value from a value. It also says functions are values, too, but you don't know what that could possibly mean, so you ignore that for now.

The syntax, (that is, the way it's written), has `\` followed by a variable name that represents what you are applying this function to. This variable can be used in the area to the right of `(->)`. It seems this lambda isn't using the `(x)` variable name it has at all. This lambda produces a new `True` whenever you apply it to any `True` or `False` value and ignores the `x` variable entirely.

This is great! It's just what you wanted. You can take your `False` coins and make new `True` ones from them! Those wily robot wizards, making such magic!

You take one of your coins, and you place it to the right of the magic coin, so its magic will bind them together. POP!

The magic coin pops, they both disappear, and a brand new `True` coin appears! Slowly you realise you just did something very silly, though.

The coin you applied to the magic coin was that single `True` coin you already had! So it did nothing, effectively. You write this down in the margin of your book, for future reference:

```
(\x -> True) True -- Equals True
-- Pointless waste. Don't do this!
```

The `--` code is called a comment, and it's not part of the code, it's just for marking notes you can write to readers of your code, including yourself. When Haskell sees `--`, it will ignore all the writing to the right of it until the end of the line.

You wrote parentheses around the lambda expression. If you didn't, Haskell would think the first `True` was a function, and that you wanted to apply it to the second `True`, and that would cause Haskell to not compile your program.

So now you've used your single `True` coin, but you got another one in its place, so no harm done you suppose to yourself, though you **did** just use up one of your magic lambda coins.

Time to try with a `False` coin. Frrrrzzzzt... POP! Brilliant! It worked. You now have two `True` coins so you can play your song. You note your new discovery in your note-book:

```
(\x -> True) False -- Equals True
```

Before you play your song though, you get curious. What if you fed one of your magically created coins into another one of those magic coins? Will it explode? Will the world stop?

Furtively, you glance over toward the robot wizard for guidance, but he appears to be engaged in an apparently amusing conversation with himself about folding wizard gowns into luggage and catamorphisms, whatever they are.

Why not try! You quickly grab a magic coin, and thrust one of your newly made coins next to it. POP! again. It does exactly what happened the first time. So you write this down, too:

```
(\x -> True) ((\x -> True) False) -- equals True
```

You read that back and it looks a little complicated. The part on the right is the coin that resulted from shoving a `False` into the magic coin. Then we put that whole bracketed thing on the right of another magic coin. It's correct, just has lots of things going on!

You slide over to the juke box, put your coins in and then begin listening to the sweet sounds of Never Gonna Give You Up by Rick Astley, a timeless classic on your home-world.

You go back to your seat and keep reading the book a bit more. You've seen definitions before. You know how to make them. We see a definition for a magic coin:

```
magicCoin :: Bool -> Bool
magicCoin = \x -> True
```

You read on and find out that because we're not using `x` in the body of our function, we can actually replace it with the underscore character; "`_`" to say that this is a function that has one argument, but it won't be used. Fascinating, you think, as you hum happily, sipping your shake.

5.2 After the Story

We leave our story here, but we see that we can use the `magicCoin` function in the same way that we used `putStrLn` earlier, except because `magicCoin` has type `Bool -> Bool`, we can feed an applied expression of it into another application of it.

```
magicCoin :: Bool -> Bool
magicCoin = \_ -> True

newCoin :: Bool
newCoin = magicCoin False

newCoinAgain :: Bool
newCoinAgain = magicCoin newCoin

newCoinAgain' :: Bool
newCoinAgain' = magicCoin (magicCoin False)
```

We can keep applying it as many times as we like, however this `magicCoin` function is only useful if you want to take a `True` or `False` and make a sure it's a `True`.

There's another way to write this function in Haskell. Let's take a

look at it:

```
magicCoin' :: Bool -> Bool
magicCoin' _ = True
```

We don't have lambda syntax here. This is regular function definition syntax. We're using "_" to pattern-match on any value at all, and not use it.

We could have also written this function like this, listing out one equality definition for each value in the argument:

```
magicCoin'' :: Bool -> Bool
magicCoin'' True  = True
magicCoin'' False = True
```

There is no reason to do this with our trivial example, but this shows you another way to write functions. We could write a function that flips a boolean value to its opposite value. This is actually built into Haskell already as the function named `not`, but we'll make our own, and show you how easy it is to read it:

```
not' :: Bool -> Bool
not' True = False
not' False = True
```

This means we first check if the input value is `True`, and if so, we return `False`. If it's `False`, we return `True`. This is called pattern matching. It's just simple pattern-matching: matching on the values.

5.3 Functions that Return Functions

Let's look at another lambda, but this one will return the `magicCoin` function! Let's really think about this. Functions are values just like `Bool` values or `String` values or any other type of values. What about the type of this function that returns a function? It will take a `Bool`, and return a `(Bool -> Bool)`, that is, it returns a function from `Bool` to `Bool`, so this is what we're looking at:

```
(\_ -> magicCoin) :: Bool -> (Bool -> Bool)
```

Let's set this up as a definition. We'll call it magicBool:

```
magicBool :: Bool -> (Bool -> Bool)
magicBool = \_ -> magicCoin
```

Those types don't need brackets, because they naturally group up to the right. This is a function that takes two arguments! Let's remove the brackets from the signature, and spell `magicCoin` out as a lambda.

```
magicBool' :: Bool -> Bool -> Bool
magicBool' = \_ -> (\_ -> True)
```

We've included parentheses for you to understand better, but they're not needed here. There are actually two other ways we could write this, let's see:

```haskell
-- using two lambdas,
-- without parentheses
magicBool'1 :: Bool -> Bool -> Bool
magicBool'1 = \_ -> \_ -> True

-- using just one lambda
magicBool'2 :: Bool -> Bool -> Bool
magicBool'2 = \_ _ -> True
```

So, how would we use this? Well, It's a function so we can just give it any `Bool` value, then it'll give us back the `magicCoin` function as a value! Then, if we want to, we can give **that** another `Bool` value, and it'll give us back the value `True`.

This might seem useless at first, but let's switch it into normal function syntax, and then we'll look at a very useful function.

```haskell
magicBool'' :: Bool -> Bool -> Bool
magicBool'' _ _ = True
```

Ok, so what if we wanted a function that tells us if any one of two values is **True**? This is a function that is built into Haskell, you'll see it later. If you're this high, or you're an adult, you can ride the scary ride!

Here's how it could be made with what we know so far:

```haskell
eitherTrue :: Bool -> Bool -> Bool
eitherTrue False False = False
eitherTrue _     _     = True
```

So if both the arguments are `False`, we should reply with `False`, however for everything else, the answer is going to be `True`. (That is: either one is `True`, or both are `True`).

Neat, isn't it? This is a very useful function, and programmers use it all the time.

This process of making a function that returns a function itself is called **currying** after the mathematician Haskell Curry (yes, that's why Haskell is named Haskell). This is how Haskell takes arguments. By doing more, we can get more than 2.

What about something more involved, like maybe adding two Integer numbers? Well, we'll show you this here just now, but it's really the topic of the next section, so don't worry if you don't quite get it yet.

We're going to define a function called `plus` that takes an `Int` value (`Int` is a whole number type), and returns a function that takes **another** `Int` value and returns the first added to the second. We're going to use the `(+)` operator / function here. Just ignore it for now other than to know that it's the way to add two numbers in Haskell.

```haskell
plus :: Int -> Int -> Int
plus x y = x + y

plus' :: Int -> Int -> Int
plus' = \x -> \y -> x + y

increment :: Int -> Int
increment = plus 1
```

```haskell
increment' :: Int -> Int
increment' = (\x -> \y -> x + y) 1

additionResult :: Int
additionResult = plus 100 25
```

First we have **plus**, which takes two arguments and returns the first added to the second (using the `(+)` operator).

Next we have `plus'` which does the exact same thing, but we're using lambda syntax, and nesting one lambda in another.

Then `increment`, which uses the plus function and gives it one of its two arguments, which as we know will give us another function back. (Which will add one to whatever you give it). After that we have the same function done with lambdas: `increment'`. Finally, we have a definition for the expression of adding 100 to 25.

5.4 Homework

This chapter might have seemed pretty simple or quite easy for you, but what we just saw is very **deep**, so it's worth thinking about some more.

Haskell functions cannot take multiple arguments, only one each. However, what they **can** do is return another function. The way we get functions to work with more than one argument in Haskell is to wrap other functions around them, as we saw above. In other words, to make functions that give functions as their result.

If each of these functions that are returned also take an argument, we can create a kind of chain of arguments. Let's see.

When we see this:

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

It's just a "sweeter", easier, more convenient way of writing this:

```haskell
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

We can put a number into `x` by applying the function to an argument, and we get a function back: `add 2` for example:

```haskell
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)

-- substitute 2 in for x.
-- note that this is not Haskell code,
-- it's just to show you what happens:
-- add 2 is equal to \2 -> (\y -> 2 + y)
-- add 2 is equal to        \y -> 2 + y
```

If you wanted, you could call this a **partial function application**, but it's **really** not what's going on. All Haskell functions only take one argument, so all we've done is supply the `add` function (which produces another function) with one argument.

When we write `add 2 3` what we're actually doing is supplying a function-producing function with one argument (`add 2`) and then immediately providing the function that gets produced by that with the `3` argument: `(add 2) 3`.

Your homework is to read through this very well, and really try to understand it. Try out all of the program fragments in this chapter, and if you don't understand any of this at all, write to us and tell us so we can adjust the book. All of the rest of your Haskell understanding relies on understanding this. We **will** explain it a little more in the next chapter, but it's quite important.

If you've already learned another programming language, then this particular point will be incredibly difficult to understand for you. If you haven't learned another programming language before, it's going to be much easier because the way Haskell works is **simpler**, in the sense that it is not as complex. Of course if you've learned something complex and you think that is ordinary, when you see something simple it makes things very difficult to understand.

You'll often hear us talking about things like "a function of two arguments" or, "put the letter 'x' in as the third argument", but this is just an easier way to say what we mean. It's not accurate, but we can use the short-hand form because we all understand that Haskell functions only take one argument, and when we say a function of two arguments, we really mean a function that produces a function that uses both their arguments in its inner body.

6 Sockets & Plugs

In this chapter, we'll explain another way that might help you to think about functions work when they're used. All of these views of functions can help you to get a rounded idea of how to use them well. We're spending so much time on this because it's the underpinning of everything in Haskell.

We could think of them like they were electronic devices, waiting to have something plugged into them. Depending on what you plug in, you will get a different result.

6.1 Reusability

Functions are one of the most basic ways to re-use expressions in a program. They save us repeating ourselves as we write programs. Functions take **variables** or **parameters** that can have different values (of a type) for each new time the function is applied.

6.2 Functions are Values

Functions are also themselves values, but they are a special kind of mapping-value from values to other values.

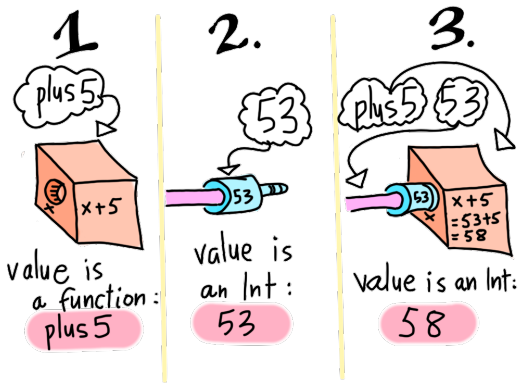
This is why `putStrLn "hi"` means to "apply the `putStrLn` function to the `String` value `"hi"`.

Another way to think of it is that `putStrLn` makes a kind of mapping between any `String` value to a corresponding `IO ()` value.

Let's imagine there's a function called `plus5` which takes a whole number as its single parameter. When we give it a value, the result is whatever its input is, but 5 more.

If we were applying this function to the number `53`, for example, we would write the function application `plus5 53`.

```
-- we apply the function plus5
-- to the number 53
plus5 53
```



Above, we show this function as a machine box with a plug and a screen that shows the "answer".

`Int` is one of Haskell's names for the type of whole numbers; both negative and positive. The mathematical name for a whole number is integer, which comes from latin and means "entire".

The function's type is written in Haskell as `Int -> Int`, which means "the type of all functions that maps from any `Int` value, to any `Int` value".

We can also say that `Int -> Int` is "an `Int` that is a function of another `Int`". Its value isn't just an `Int` until you "give" it an `Int`.

6.3 Plugging Values into Functions

So in the graphic, we "plug" `53` into this `plus5` box, and it displays `58`. Comparing our graphic with the way **function application** is actually written, we can see it's reversed. In Haskell, as well as in Math, we usually plug values in to the **right** of the function, (or sometimes, with operators, the left and right), whereas in our box graphic, or with audio equipment, we're plugging values in on the left of the boxes (which represent functions).

This is an important point. The way people think and do things is often put a different way around compared to the way you have to write things in Haskell. This is often the same thing with Math. In Math and Haskell, you have to be more consistent and precise. For example, in everyday life, we'd say "add five to three", but in Math and Haskell, we'd write `5 + 3`.

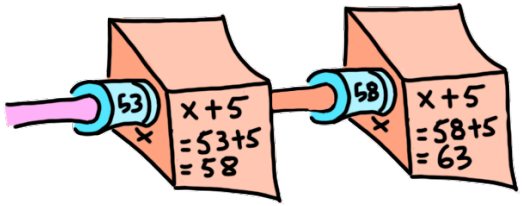
It's good to realise there are usually many ways to look at some-

thing. For example, if we look at a toggle switch, and it's marked "off", does that mean it's currently off, or that pressing it will "action" the off functionality? (ie turn it off). There is no one true right answer, so it's good to be aware there are many ways to express the same thing that can often appear as complete opposites to each other.

Getting back to our machine, the moment we plug our `53 :: Int` into the box, the type of the box changes from `Int -> Int` to just `Int`. We can then plug that whole expression `(plus5 53)` into any other function box that takes an `Int` as an argument. We may have to use brackets, though, in Haskell.

```
-- apply plus5 to 53:
plus5 53

-- apply plus5 to
-- the application of plus5 to 6
plus5 (plus5 6)
```

What if we wanted to plug the expression `plus5 53`, into the function `plus5` again? Well, we could do that like this: `plus5 (plus5 53)`, and the result would be `63`:



We can clearly see that the "output" of the first box as 58 can be plugged into the "input" of the second box. (Don't get confused, this is not an `IO` action! We're not talking about actually outputting these numbers on the screen or anything, just plugging values into functions as a metaphor).

6.4 Defining Functions

So, let's move on and read a definition for this `plus5` function. We already know what it does:

```
plus5 :: Int -> Int
plus5 x = x + 5
```

This is the type declaration and definition for a function that takes one argument of type `Int`. We're naming that argument `x` here in this definition (that's why the `x` appears on the left of the `=` sign). We could have named it almost anything we liked. To "use" this function, you supply `plus5` with an `Int` value by placing it to the right of it like this: `plus5 7`.

The definition uses something special we haven't seen yet called an **operator**. Here it's an **infix** function called `(+)` that is named as the `+` symbol, and it takes two numeric arguments, one on either side! A function is called an **infix** function when it appears between its arguments. Normal functions are called **prefix** because they are placed before their argument(s). Functions that are named

as symbols like `(+)` here are called **operators**, and they almost always only ever take exactly two arguments, and are usually infix, like `(+)`. You'll also notice when we talk about them, we put parentheses around them. In Haskell this is how they're referred to outside of when you're using them in a function application.

Let's look at another program that does almost the same thing. Notice we're using a different variable name here (we called it number), rather than `x`.

```
plus6 :: Int -> Int
plus6 number = number + 6
```

Now we'll present another way to write this. If you have an infix operator such as `(+)`, and you want to use it as a prefix function, you can just wrap it in parentheses. This function works exactly the same as `plus6`:

```
plus6' :: Int -> Int
plus6' number = (+) number 6
```

6.5 Operator Sections

Next we'll present **another** identical function, but using what's called a **section**. A section is a partially applied operator. That means it has one of its two arguments supplied, and that becomes a function. It always uses round brackets.

```
plus6'' :: Int -> Int
plus6'' number = (+6) number
```

See if you can guess the type of `(+6)` right now. First, you might want to think about the type of the `(+)` operator. It takes two numeric arguments, and returns one numeric argument. So, if one of its arguments are supplied, it will become a function of only one argument. Then, we apply this function to the number variable to get our result.

So, you can think of the type of the function `(+6)` as taking a single number, then returning a number.

It doesn't matter which side you put the value on with the operator `(+)`, because `(+)` takes two identical arguments, and is an operation that works the same no matter the order. In math, this property is called the **commutative property**.

6.6 What is commutivity?

The word commute can be broken into **com-**, which means altogether, and **mut-** which means to change. Commutativity means the ability to interchange, so we can see that we can interchange the numbers between either side of `(+)`, and it makes no difference.

Here are four functions that are identical in result:

```
sevenPlus :: Int -> Int
sevenPlus number = (7+) number
```

```haskell
sevenPlus' :: Int -> Int
sevenPlus' = (7+)

plusSeven :: Int -> Int
plusSeven number = (+7) number

plusSeven' :: Int -> Int
plusSeven' = (+7)
```

6.7 Homework

Your homework is to get familiar with more definitions, and seeing how function arguments are used in function bodies. Don't get too worried or confused by the many tricky weird looking things you'll see as you look at other code.

We'll show you some code below. You should also do an internet search for "haskell defining functions", go through the first 10 or so returned pages and quickly scan through them for value and function definitions. Rememeber there are two ways to define functions: either with the lambda syntax like `sevenPlus = \number -> (7+) number` or with the "normal" syntax like `sevenPlus number = (7+) number`. Your aim here is simply to recognise where the definitions are.

Here are the examples. Remember, don't get caught on what you **don't** know, just look for what you **do** know. Remember, you're just looking for the definitions, especially the function definitions:

```haskell
squaredNum :: Integer -> Integer
squaredNum x = x ^ 2

lengthNum :: Show a => a -> Int
lengthNum n = length $ show n

bool1, bool2 :: Bool
bool1 = True
bool2 = False

notBool :: Bool -> Bool
notBool b = if b == bool1 then bool2 else bool1

veryNotBool :: Bool -> Bool
veryNotBool = \aBool -> notBool aBool

sumFrom1To :: Integral a => a -> a
sumFrom1To 0 = 0
sumFrom1To n = n + sumFrom1To (n - 1)

isEven :: Integral a => a -> Bool
isEven n = n `mod` 2 == 0
```

7 Output Other Things

We've seen `putStrLn`. It lets us output any `String` on the screen.

What if we want to print out a number instead of a `String`?

We have the following definition for an expression that is adding two numbers, and we want to print the resulting value out in a program.

7.1 Integer or Int?

```
number :: Integer
number = 100390 + 29389
```

(`Integer`, by the way, is the type of **unbounded whole numbers** in Haskell. Unbounded means they have no upper or lower limit (or bounds). In contrast, the `maxBound` of `Int` that this document is being prepared on now is 9223372036854775807.)

So, we know that the `putStrLn :: String -> IO ()` function takes a `String`, and leaves us with an `IO ()` value.

We don't have a `String`, though, we have an `Integer` value. How can we match these up so we can print out our number on the screen?

In order to answer that, let's first look at the type of the `(+)` function. We know from experience that it takes a number and another number and returns their sum as a number, but the actual type isn't something we've seen, and includes some special new stuff for us. Let's look:

7.2 Type Variables

```
(+) :: Num a => a -> a -> a
```

Ok, breathe. Let's first look at the right side: `a -> a -> a`. Why all the `a`'s?

Well, we know from all the `(->)`'s that this means it's a function that takes **two** values of type "`a`", and returns a third "`a`". What is "`a`", though? It's what's called a **type variable**. That means it can be any type. If it starts with a lowercase letter, it's a type variable. If it starts with a capital, it's an actual type, or a typeclass, which we'll explain soon (Num is a typeclass in `Num a => a`).

One important point about **type variables** is that while they can be any type at all, all the "`a`"s must still be the same type as each other when using the function! So if we used an `Integer` value as our first argument, so saying that the first type `a` was `Integer` in `(+)`, then we would need to use an `Integer` as our second argument, too:

```
-- both types must be the same,
-- so this will be fine:
goodNumber = (3 :: Integer) + (5 :: Integer)
```

```haskell
-- this would cause a type error
willNotWork = (3 :: Int) + (5 :: Integer)
```

If we don't specify the type of our numbers, Haskell's **type inference** works it out for us, which saves a lot of time and hassle.

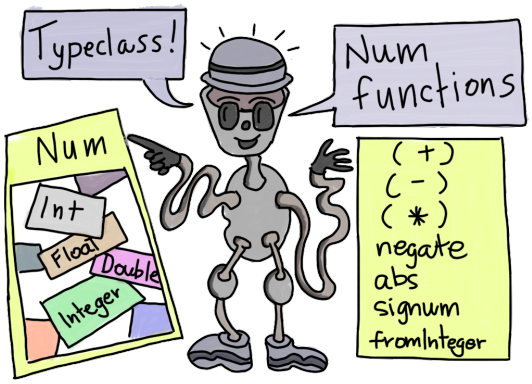
7.3 Type Variables can be named anything

Ok, so here is another way to write the `(+)` function, which shows you that variables don't necessarily have to be named `a`:

```haskell
-- it's a lot smaller to write
-- 'a' than 'theNumber'
(+) :: Num theNumber =>
       theNumber ->
       theNumber ->
       theNumber
```

7.4 Typeclasses

Now we have to think about the "`Num a =>`" part. That can be read as "`a` is constrained to types which are instances of the `Num` typeclass". This mouthful means that the `(+)` function can take two arguments of any type at all (here we're naming them `a`), as long as that type (again, here called `a`) is an instance of a typeclass called `Num`. Luckily for us, all numbers are!



A **typeclass** is not a concrete type like `Integer`, `Int` or `String`. It's a way of tagging **many** types (a "class" of them, if you like) so that we can have functions or values that work with many similar types that do similar things, but that are actually different. Let's take a look:

```haskell
-- a small int 5:
intFive = 5 :: Int

-- a "floating-point" value of 10.3
floatTenPointThree = 10.3 :: Float

-- add them together with (+). This will not work...
-- because both types are concretized (or specialized)
errorResult = intFive + floatTenPointThree

-- add them together with (+). This will work
result = (fromIntegral intFive) + floatTenPointThree

-- the result is 15.3
```

Here we're using `fromIntegral` to build an "unspecialised" `Num a => a` version of the `Int` value of `intFive` so we can subsequently add it to `floatTenPointThree`. The value `5 :: Int` is not of type `Float`, so the types won't match unless we do this. However, if either of the values are of type `Num a => a`, then `(+)` will typecheck because it can match both types together (by concretizing the `Num a => a` type to `Float`).

The `Float`, `Int`, and `Integer` types are all instances of the `Num` typeclass. There are many numeric types in Haskell such as these. To be able to do arithmetic functions on different numeric typed values, they are tagged as `Num` which allows us to define each of the simple arithmetic functions for each type differently, but use them all with the same name and interchange values.

This "tagging" is called making a type an instance of a typeclass. When a programmer does this, they provide a definition against the particular type, for the functions that the typeclass requires.

So we can see that `(+)` can add a `Float` or an `Integer` to a `Num a => a` without a problem. The `Num` typeclass is, in this way, like a kind of contract that programmers of a type can decide to kind of "subscribe to" which gives them the ability to write implementations of the functions that the `Num` typeclass provides. In turn, the typeclass system gives that type the ability to work with all the other types that are instances of that typeclass.

So the `Num` typeclass means there actually isn't only one definition for the functions for addition: `(+)`, subtraction: `(-)`, multiplication: `(*)`, negation: `negate`, etc but rather that each type — that is, each instance of `Num` — has **its own** definition for each of these functions.

7.5 The Show Typeclass

What does all of this have to do with printing our number on the screen? Remeber, that problem we started the chapter with?

Well, there's a typeclass called `Show` (with a big S), and this provides a single function: `show` (with a small s), that can take any instance of `Show`, and makes a `String` version of it. Let's look at the type of the `show` function:

```
-- takes a "showable" thing
-- and returns a String
show :: Show a => a -> String
```

We see that show is a function which takes a single argument of any type (the "`a`" type variable above) constrained to the `Show` typeclass, and returns a `String`. That single argument is anything that has an instance of `Show` defined for it. We know this because of the `Show a =>` constraint.

Printing things to the screen is such a common thing to do that many types have an instance of `Show` already, including of course, `Integer` and `Int`, so getting back to the first program of this chapter, we can just apply `show` to our `Integer` and then pass it to `putStrLn`. Let's see how:

```
number :: Integer
number = 100390 + 29389

main :: IO ()
main = putStrLn (show number)
```

7.6 Parentheses and Precedence

Parentheses are needed on `show number` because `putStrLn` only takes **one** argument, and the function application **precedence** rules mean that taking them off would give it **two**. Precedence is a fancy-pants word that simply means "which things come before or after which other things". If we left off the parentheses, we would have this: `putStrLn show number`, which Haskell would see as "apply `putStrLn` to the value `show`, and then apply that to the value `number`". However, `putStrLn` takes only `String` values, and `show` is a function, so that would **definitely** be a type error.

7.7 The print Function

We have one last trick up our sleeves to show you (pardon the pun). It's the `print` function. This is very similar to `putStrLn :: String -> IO ()`, but rather than taking a String, it can take a value whose type is any instance of `Show`! Let's see a version of our program that uses `print :: Show a => a -> IO ()` rather than `putStrLn`:

```
number :: Integer
number = 100390 + 29389

main :: IO ()
main = print number
```

7.8 Homework

See if you can work out what the following program does.

```
number1 :: Num a => a
number1 = 1 + 5 + 7 + 3 + 2

number2 :: Num a => a
number2 = number1 * number1

main :: IO ()
main = print number2
```

Hint: don't get caught up by the types of the values. This **will** probably be confusing, and **should** confuse you at least a little bit. We'll explain what's going on later, however, the important thing is just see if you can work out what the program will **do** when you run it.

8 Make Decisions

In this chapter we'll look at all the ways we can make code that gives results that depend on values.

8.1 if...then...else expressions

First up, let's look at a program that takes a name, then prints a message, depending on what the name is.

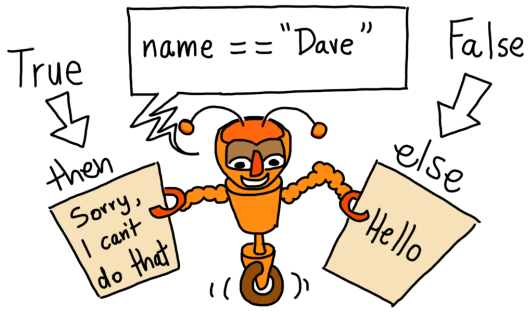
```
message :: String -> String
message name = if name == "Dave"
               then "I can't do that."
               else "Hello."

main :: IO ()
main = putStrLn (message "Dave")
```

We see `message` is a function that takes a `String` and returns a `String`. The `String` it takes as an argument is named `name` in the body of the function. It's **pattern-matched** into the `name` variable, is another way to say this.

The `if...then...else` expression is one way we can control what will happen in Haskell programs.

It's very good for what's called a **two-way branch**; that is, when there are two "paths" to take: the path for `True` or the path for `False`. (It's either `True` that `name` is `"Dave"`, or it's `False`).

The result of the `if` expression is dependent on the variable named `name`.



So what about this `(==)` operator that we saw in that program? Let's look at its type:

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

Here we have a new **typeclass** constraint, this time called `Eq`. This operator works out if its two arguments evaluate to the same value. If so, it returns the `Bool` value `True`, otherwise `False`.

`Bool` is the type which comprises just the values `True` and `False`. These are used to express whether things are true or not in Haskell.

`Eq` provides the `(==)` and `(/=)` operators. They mean "is equal to", and "is not equal to" respectively. `Eq` is short for equal or equality.

An `if` expression has three sections, and it must **always** have three sections. The first section is an expression that must evaluate to a `Bool` value. You can begin to see why `(==)` is quite an important operator now, can't you? When it evaluates to `True`, the expression that follows "then" is returned, otherwise the expression after the "else" is returned.

The entire `if` expression always results in a single value, both of its return expressions **must** have the same type.

8.2 Nesting if Expressions

So, what if we want to actually test for more than just two alternatives? (maybe "Dave", "Sam" or other).

Well we can "chain" these `if` expressions by putting another one into the first one:

```
message :: String -> String
message name =
  if name == "Dave"
  then "I can't do that."
  else if name == "Sam"
       then "Play it again."
       else "Hello."
```

So now, it'll check if the name is Dave: if so, it'll respond with "I can't do that" as before, otherwise if the name is Sam, it'll respond with "Play it again", and if it's neither, it'll be "Hello". Phew! Look at that if expression! What a mouthful. And this will only get more annoying as we add more options.

Let's see the same program but using the `(/=)` operator instead:

```
message :: String -> String
message name =
  if name /= "Dave"
  then if name == "Sam"
       then "Play it again."
       else "Hello."
  else "I can't do that."
```

No surprises here, we just have to flip the branches around as we've done.

Having this many branches in our `if` expressions is not very easy to read. Let's look at a better way to do the same thing. Before we do this, though, it should be mentioned if you're writing these programs in, then realise that the **spacing matters in Haskell**! So, we must indent our lines properly. There is actually another way we can write haskell which uses lots of punctuation instead of spacing, but spacing looks nicer, so we will use that.

8.3 Case Expressions

Here we're using a **case expression**. You can see how it works pretty easily when comparing it to the nested if expressions from our previous example.

```
message :: String -> String
message name =
  case name of
    "Dave" -> "I can't do that."
    "Sam"  -> "Play it again."
    _      -> "Hello."

main :: IO ()
main = putStrLn (message "Dave")
```

To evaluate a `case` expression, the expression between "`case`" and "`of`" is first evaluated, then Haskell will run through all the patterns we have given it on the left of the `->` symbols, and try to pattern-match the value with them. If it finds a match, it returns the corresponding expression to the right of the `->` symbol.

`Case` expressions are incredibly powerful because of the pattern matching we can do. Here we've just shown you an extremely basic example where the single name expression `name` is matched to simple value `String` patterns.

What about the underscore (`_`) pattern? This pattern matches everything in Haskell, and it's included to make sure any time our function is called in the future with something we didn't anticipate, it will still work. Notice that the order matters. Dave will be matched before Sam. In this example it's not so important. Take a look at the following example, though:

```
message :: String -> String
message name =
  case name of
    _      -> "Hello."
    "Dave" -> "I can't do that."
    "Sam"  -> "Play it again."

main :: IO ()
main = putStrLn (message "Dave")
```

The order matters! In this case, even if `name` is `"Dave"`, the code will never get that far, because the underscore matches on everything, and it's first in the list!

In this case, our program will compile just fine, but it's not what we want. This is called a **logical error**, because while it's syntactically correct, it doesn't have the correct logic. If we compile this with a Haskell compiler such as GHC, it will issue us a **pattern-match overlap** warning, letting us know that we've got multiple paths of logic flow for the same inputs.

Do note, also, that all of the types of the result expressions have to be the same. The same rule applies from the `if` expressions, above. You can't have a different result type in any of the expressions on the right. The whole `case` expression is a single expression, so it must result in a value of a single type.

8.4 Guard Patterns

Let's look at **yet another way** to do the same thing, this time using what's called a **guard pattern**:

```
message :: String -> String
message name
  | name == "Dave" = "I can't do that."
  | name == "Sam"  = "Play it again."
  | otherwise      = "Hello."
```

So we immediately notice that the function definition's = symbol is gone from the right hand side. It no longer says `message name =` ... but rather there are multiple "definitions" each with a pipe (|) symbol in front of them.

The way this works is that the expressions on the left are tested for equality to `True`, in order. When a `True` value is found, it returns the expression to the right of the = sign that corresponds to that expression. This is not pattern matching like in the case expression, but rather test for truth, so it's subtlely different. This is quite good for when you to test for several things.

Also notice that where a `case` or `if` expression can actually be inserted anywhere you like, this one is only usable in named function definitions. You can't use this form within a lambda, for example.

You can also see that we're using something called `otherwise` here as our default expression value. The `otherwise` identifier is defined very simple, as `True`, so we could have written this:

```
message :: String -> String
message name
  | name == "Dave" = "I can't do that."
  | name == "Sam"  = "Play it again."
  | True           = "Hello."
```

Which of course, because it's testing for `True`, will always match. The only difference is, `otherwise` actually means something to human programmers, so that's why we use it.

Wherever possible, we should endeavour to make our programs as clear as possible for our future selves, and others who may want to read our code. Sometimes it's quite literally the difference between our code being used or not.

8.5 Argument Pattern Matching

There's still one more way we can write this, so let's see that now. This is just a simple regular function definition, but with many definitions, and using pattern matching on values in the argument list:

```
message :: String -> String
message "Dave" = "I can't do that."
message "Sam"  = "Play it again."
message _      = "Hello."

main :: IO ()
main = putStrLn (message "Dave")
```

Here we're using pattern matching again, but directly on the argument list. Notice that the `name` variable is gone entirely, and our old friend the underscore is back. That's because we're directly pattern matching on what the `name` value would be.

8.6 Conclusions

In the simple examples we've shown, none of the techniques stand out as obviously better or worse, except the **if expression** when we want to match on more than one thing.

The **case expression** is probably the best fit, because there's less repetition. Each have benefits and drawbacks, and we'll see more of each of them as we proceed. This section is mostly to get you familiar with them.

As what we need changes, the different **conditionals** will make more or less sense. What if we needed to detect if the name started with a "D"? In that case, the **if expression** or **guard pattern** would make more sense to use than the others. The **case expression**, or other direct pattern matching style examples wouldn't make sense there.

Let's finish by adding another simple clause to our case expression example:

```
message :: String -> String
message name =
  case name of
    "Dave"  -> "I can't do that."
    "Sam"   -> "Play it again."
    "Creep" -> "Feeling lucky?"
    _       -> "Hello."

main :: IO ()
main = putStrLn (message "Dave")
```

8.7 Homework

Two parts to your homework. First is to try all the examples yourself. Experiment with changing `"Dave"` to other names, then do an internet search for examples of each type of conditional technique and recognise the pieces.

Don't get too worried that they won't look as simple as our examples here. Just use our examples as a kind of guide, and try to pick out the pieces you **do** recognise, and don't get confused if the examples you find look crazy. Ignore the crazy for now and look for the parts you **do** recognise.

The second part of your homework is to help reinforce your understanding of **currying**. Maybe you forgot what that was. It's when we use two or more functions wrapped around themselves to make a way for a function to take multiple arguments.

Let's look at a function that might look a little bit strange at first:

```
addThem :: Int -> Int -> Int -> Int -> Int
addThem a b c d = a + b + c + d
```

You can probably work out that this function adds its four arguments together. Of course, what we really mean is that it has the **effect** of having four arguments and adding them together, but we understand what is **really** happening when we define a function like this. What is really happening is that Haskell builds a "multi-level" function that looks like the following (notice the indentation and

how each function argument lines up with an `Int` in the type signature):

```
addThem :: Int -> (Int -> (Int -> (Int -> Int)))
addthem =  \a  -> (\b  -> (\c  -> (\d  -> a + b + c + d)))
```

And, because the way function application works in Haskell, we can apply values to it in this way `addThem 1 2 3 4`, then it will return `10`.

So, we can see from this that `addThem` is defined as a function that takes a variable called `a` and returns a function. **That** function uses `a` deep inside its bowels; that second function is a function that takes a variable called `b`, and so on, until you get to the inner function body of the function that takes `d` as its argument. You can see how these variables match up to the type signature above it, which we've put parentheses around to group them, showing the way Haskell builds the functions up more clearly.

Well, to unpack this a little bit more, let's make five different definitions to show each step of this process, and provide you with type annotations of each of those definitions. First up, we'll create a definition that where we apply `1` to the `a` variable, and so we can remove its outer function wrapper, the one with the `a` variable in it:

```
-- here we have applied the addThem function to 1
addThemOne :: Int -> Int -> Int -> Int
addThemOne = addThem 1
-- which looks like this:
-- addThem 1 == \b -> (\c -> (\d -> 1 + b + c + d))
-- this is the same thing as applying the value 1
-- to that function / lambda:
-- (\a -> (\b -> (\c -> (\d -> a + b + c + d)))) 1
```

We can see from that quite clearly that `1` has been substituted in for `a` within the function. We don't need the outer part of the function syntax `\a ->` any more because we've applied that function to the value `1`, which makes it disappear.

Next we'll satisfy the `b` variable with the value `2`, and we can therefore also similarly remove **that** function wrapper, because it has been applied:

```
-- here we have injected 2 into the addThemOne function
addThemOneToTwo :: Int -> Int -> Int
addThemOneToTwo = addThemOne 2
-- which is the same thing as
-- addThem 1 2
-- and would look like this:
-- addThemOne 2 == (\c -> (\d -> 1 + 2 + c + d)
-- which is the same thing as this
-- function application, ie applying 2 to it:
-- (b -> (\c -> (\d -> 1 + b + c + d)) 2
```

Next we will place `3` into the `c` variable. Are you noticing the type signatures? They're getting smaller by one with each application of our function. This should make sense because each time we're building a new function by applying a value to a previous function which produces a function of one less argument. Let's continue:

```
-- here we have applied the addThemOneToTwo
-- function to the value 3
```

```
addThemOneToThree :: Int -> Int
addThemOneToThree = addThemOneToTwo 3
-- which is the same thing as
-- addThem 1 2 3
-- or addThemOne 2 3
-- and would look like this:
-- addThemOneToTwo 3 == (\d -> 1 + 2 + 3 + d)
-- again, this is the same as this
-- function application:
-- (\c -> (\d -> 1 + 2 + c + d)) 3
```

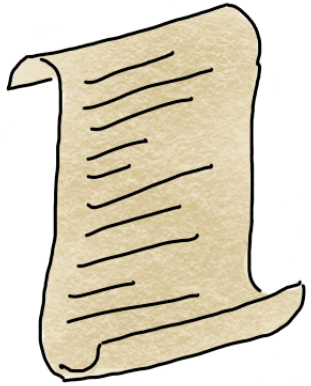
Finally, we put the value `4` into `d`:

```
-- here we take 4 and put it into
-- the addThemOneToThree function
addThemOneToFour :: Int
addThemOneToFour = addThemOneToThree 4
-- which is the same thing as
-- addThem 1 2 3 4 or
-- addThemOne 2 3 4 or
-- addThemOneToTwo 3 4 or
-- and would look like this:
-- addThemOneToThree 4 == 1 + 2 + 3 + 4
-- which equals 10
-- again, this is the same as this
-- function application:
-- (\d -> 1 + 2 + 3 + d) 4
```

We hope this is clear, if it's not at all, we'd really appreciate your feedback. You can give it by following the feedback link on the main page.

9 Shop For Food with List

We're going to use Haskell to make a shopping list for when we go to shop for food, and to write some code to find out how many items we have on it.



9.1 The Smallest List

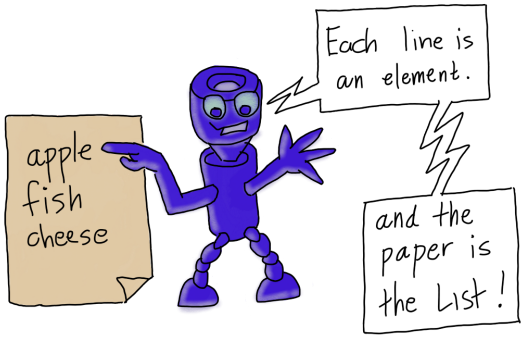
Before we make the list, let's see how to make the simplest list possible in Haskell, calling it `aList`:

```
aList = []
```

We'll get to the types in a minute, but this is a definition for a name called `aList` and it's defined to be the **empty list**. This is the name

given to those square brackets on the right with nothing between them: it's a list that has nothing inside it.

What about a list with one thing inside it? Let's say we want to have a list with "sauce" written in it. Well, there are two main ways to do that. First up, we could build it up using the (:) operator:



9.2 The (:) Operator

```
aList2 = "sauce" : []
```

This might look a bit weird, but the `(:)` operator is just a function that makes new bigger lists out of old ones. It's an **infix function** and like all operators, it takes two arguments: in this case, an element (`"sauce"`) and a list (`[]`). Remember that **infix** means it sits in between its arguments? What `(:)` does is make a new list with the **element argument** prepended to the **list argument**. That is, it returns a fresh list with the element at the front of the list.

It's important to notice that `"sauce" : []` doesn't **do** anything to the `[]` value. It's still the empty list, and it doesn't change when we write expressions that use it. The `aList2` definition and expression just **uses** it, but it doesn't **change** it.

Because `(:)` is an operator, we can also apply it as if it were a prefix function, by using parentheses:

```
-- a list with just pie on it
aList3 = (:) "pie" []
```

This is the same thing as `aList2`, but with `"pie"` instead of `"sauce"`. It's a different **syntax**, though. Syntax is the form of writing something. If we use parentheses around an operator, it works like a regular prefix function of two arguments.

9.3 List Syntax

There's yet **another** way to make a list, which is the most usual way that you'll see in Haskell programs:

```
-- a list with just pie on it
aList4 = ["napkin"]
```

You need to recognise these three ways a list can be written, but we'll give you lots of practice, so don't worry!

9.4 The List Type

So what's the type of aList4? Let's see:

```
aList4 :: [String]
aList4 = ["napkin"]
```

Ok so, usefully, its type looks very similar to how we write its values!

There is another way to write this type (it's a bit of a strange way to write it, though), so let's quickly see that here:

```
aList4 :: [] String
aList4 = ["napkin"]
```

We can put the type name (the brackets) on the left, or wrapped around the second type.

So, because it "takes another type" as an argument similarly to a function, we can call the list type a kind of container type: it has two pieces - the container type (List), and its element type in the case of List (in the above, it's String). That is, the list type is a **parameterised** type.

There are many container types in Haskell, and almost none of them besides this one have this "wrap-around-another-type" syntax. This can sometimes confuse people when they get to learning the other container types, which is why we're introducing the normal syntax now.

Just know that even though list has a special syntax, it's still a regular data type, which is why we can write the string list type using regular type syntax as `[] String` as well as the special list syntax of `[String]`.

9.5 Lists of Other Types

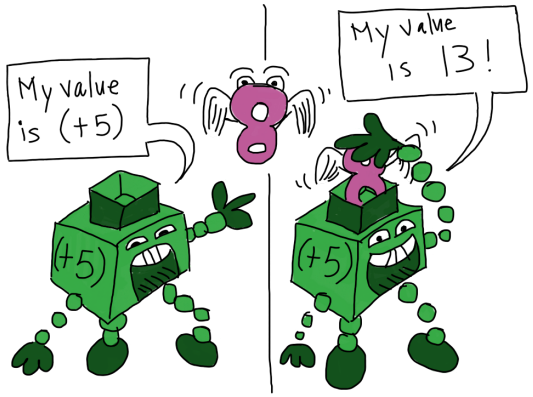
So, moving on... what would a list with an `Integer` value in it look like?

```
aList5 :: [Integer]
aList5 = [879]
```

Ok. So by now you've probably noticed how using the list type is sort of similar to function application, but for types rather than values.

A function is a thing that "wraps" another value to turn it into another value. The function `(+5)` for example, wraps a number to

turn it into a number 5 greater than the first one: we do this by applying the function to the value.



Well, these "container types" like list, they actually take a variable, too! This is obviously not like the **value variables** that functions take, though, it's a **type variable**! This means in a type signature, you put another type with it as we've seen, which will result in a composed type. In the case of list, the parameter is the **type of its elements**. By itself, List doesn't actually mean anything other than the potential for a list-like type of some variety.

That is, by itself, in the same way that functions don't result in a concrete value until you apply them to another value, types with type variables are not a concrete type until you put a type with them.

By the way, we can, of course, write the above code using the `(:)` data-constructor operator instead. The following means the exact same thing:

```
aList6 :: [Integer]
aList6 = 879 : []
```

9.6 Lists with More Items

So this is great and all, but we want to write a whole shopping list, not just a list with one item on it! We're going to need more items.

Here's a definition for a list with two items:

```
aList7 :: Num a => [a]
aList7 = [1,3]
```

We notice two things here. First, to have more items you put commas between items in this syntax. Second, we notice that we haven't told Haskell a concrete type for our list. It's a `Num`-constrained **value** called "`a`"!

9.7 Polymorphic Values and Types

This means that we're letting Haskell work out what type the numbers are. All we care about is that the type that Haskell picks for them is an instance of the Num typeclass. This is not to say that they don't have an actual type when Haskell compiles our code, it definitely will.

These kinds of values with typeclass-constrained types, like the type `Num a => a`, are called **polymorphic values**. Polymorphic just means many-shaped, and comes to us from Greek. The empty list (`[]`) is actually a polymorphic value, too. That's how we're able to write `"sauce" : []` or `1 : []` and have Haskell still match the types properly:

```
-- the type of the empty list.
-- "t" could be any variable name
[] :: [t]
```

So the empty list is a value, which from looking at its type signature, can see we use any type we like with it. When we hook it up to a value with a more concrete type like `Integer`, say, Haskell infers we must mean the empty list of `Integer`, because nothing else would make sense (`[] :: [Integer]`).

Interestingly, the List type is a **polymorphic type**. That means it takes an argument, which will be another type! We just saw this a few times when we saw Lists of String (`[String]`), and Lists of Integer (`[Integer]`), and just now a List of `Num a => a` when we saw `aList7 = [1,3] :: Num a => [a]`.

9.8 The (:) Operator Again, Binding & Associativity

Okay, so here's another definition for this same list, constructed using the `(:)` operator:

```
aList8 :: Num a => [a]
aList8 = 1 : 3 : []
```

There's nothing much new here, except that if you've got a mind like a super keen blade, you'll have noticed something odd about the `(:)` operator, which is in order for this code to work, Haskell must be applying the `(:)` function to `3` and `[]` first, otherwise it would be applying it to `1` and `3`, which we know is impossible because of the type of `(:) :: a -> [a] -> [a]` — its second argument has to be a list.

So what's going on here, then? The `(:)` operator is **binding** to the right; it has what's called **right-associativity**. Binding can be described as when you apply a function to a value or variable, you're binding the value as the function's argument. **Associativity** is just a fancy name meaning "the order functions evaluate their arguments in when there are no parentheses around".

The `(:)` operator usually functions as though it were written like this:

```
-- we don't need these parentheses
aList8 :: Num a => [a]
aList8 = 1 : (3 : [])
```

Interesting! Most functions we've seen so far bind to the left (they're left-associative). So this one is right-associative, in other words, it prefers to associate to the things to the right of it first before it gets to the left. So Haskell looks at the `1`, then looks at the

first `(:)` and says "hey, let's wait until we've seen what's more to the right of this `(:)` before we do any binding or application here".

It's good that it's set up like this, because if it were left-associative, then it would try to do `(1 : 3)` first, which would not work because `(:)` takes an element and a list as arguments, and `3` is not a list. In that case, we'd have to write lots of parentheses to get it to work properly, which would be a pain.

Again, here's the type of the `(:)` operator:

```
(:) :: a -> [a] -> [a]
```

This means it's a function that takes any value at all of some type called "a", and a list of that same "a" type, and returns another list of that type.

9.9 The Shopping List

We've covered a lot so far, so let's just take a look at the shopping list program that prints a shopping list on the screen while we catch our breath.

```
shoppingList :: [String]
shoppingList =
  [ "Carrots"
  , "Oats"
  , "Butter"
  , "Apples"
  , "Milk"
  , "Cereal"
  , "Chocolate"
  , "Bananas"
  , "Broccoli"
  ]

main :: IO ()
main = print shoppingList
```

This program just prints the shopping list data using `print`, which uses the built in `Show` instance for `List` and prints a list on the screen in the usual program notation above. It's not very pretty!

How do we know it uses `show` to print this out? Well the `print` function takes a value whose type has a Show instance, remember? Let's remember its type: `print :: Show a => a -> IO ()`.

9.10 Counting The Items

What if we want to print out the number of items on the list instead? Well there's a function called length that will give us just that:

```
length :: Foldable t => t a -> Int
```

This type signature looks kind of crazy because the function is **extremely** general — that is, it works with values from a whole **class** of **container types**! Let's break it down.

We recognise there is a **type constraint** `Foldable t =>`, but that it's using a typeclass that is new for us called `Foldable`. This is specified on the type variable "`t`". Then there's a type variable "`a`", which is completely unconstrained, so it can be anything we like.

This "`t a`" might feel a little bit familiar, because it's a generalised version of what we've just seen when we arrange our list's type in that **odd** way we discussed: `["hey"] :: [] String`. Let's see, does this look similar?

```
aList9 :: [] String
aList9 = ["Cat Food", "Lasagne"]
```

What about if we told you that list **is** an instance of the `Foldable` typeclass? Well, it is. So, the type `[] String` could match `Foldable t => t String`.

So, thinking about the `length` function again, it takes a single value. The type of that value is "`t a`" where "`t`" is a wrappering type around "`a`", and where `t` is constrained to types that are `Foldable`.

`Foldable` is a class of types that have a shape or structure that lets us reduce them to a single value in various ways. The `length` function reduces a `Foldable` structure down to a single `Int` value representing the item count of the "container of items". You can see why it's called `Foldable` if you think of cooking where you fold eggs into batter — you might fold three eggs, say, into one single batter mix.

The fact that the list type has an instance specified for `Foldable` means it can be the "`t`" in our type signature for `Foldable t => t a`. We know from the above that when we write the type `[String]`, it means the same as the type `[] String`.

If this is at all confusing, well don't worry about it too much for now. We'll see many more examples of types that are composed of two or more parts like this later on.

So, we can safely pass `length` our `shoppingList`, because the types will match. It will happily return the number of items in it as an `Int` value.

9.11 Adding a Message with (++)

Let's adjust our program so it prints a nice message with the number of items in our shopping list:

```
shoppingList :: [String]
shoppingList =
  [ "Carrots"
  , "Oats"
  , "Butter"
  , "Apples"
  , "Milk"
  , "Cereal"
  , "Chocolate"
  , "Bananas"
  , "Broccoli"
  ]

main :: IO ()
```

```
main = putStrLn ("There are "
                 ++ (show (length shoppingList))
                 ++ " items on the shopping list.")
```

Wow, that is a **lot** of parentheses. Later we'll see how to reduce the number of parentheses used. Also, what is this new operator `(++)`?

Well, `(++)` joins, or **concatenates**, two lists together. Happily for us, the `String` type is itself just a list of type `Char` (the type of all written characters). That is, `String` is identical to `[Char]`. The type of `(++)` is `[a] -> [a] -> [a]`, so here we're just joining three strings together to print out.

Now, let's look at another way to represent `shoppingList` (which Haskell sees as identical to the above):

```
shoppingList :: [String]
shoppingList =
  "Carrots" :
  "Oats" :
  "Butter" :
  "Apples" :
  "Milk" :
  "Cereal" :
  "Chocolate" :
  "Bananas" :
  "Broccoli" : []
```

This shouldn't pose too much of a problem by now. As we know, the `(:)` operator has type `a -> [a] -> [a]`, which means it takes a single item of any type, and a list of items of that same type, and then returns a list of items with that same type.

We can do like the above; chain the function applications of (:) to make a list, as long as we put a list of some kind at the end.

9.12 Pattern-Matching with the (:) Value Constructor

The `(:)` operator is very handy. Because it's a **value constructor**, we can also use it in pattern-matching to match parts of lists in arguments to functions! This is the case for **all** value constructors.

Let's take a look at a function that will get the first item from any list of strings (which includes our `shoppingList` obviously, because it's just a list of strings). If the list is empty, it'll just return a blank `String`:

```
firstOrEmpty :: [String] -> String
firstOrEmpty []     = ""
firstOrEmpty (x:_)  = x
```

Remember when pattern-matching, the definitions or patterns that appear earlier in the list will get their arguments matched first, if they can.

9.13 Totality and More on Pattern-Matching with (:)

The first definition for this function is very easy to understand. It just matches on the empty list, which if it finds it, it returns the empty `String`. We have this in case an empty list is passed in as

its argument. If we left it out and an empty list was passed, it would cause a runtime error when we ran our program. This would be bad. Including it makes the function **total**, which means it covers all possible values.

It's good to have total functions because programs change, and when they do, unexpected things can happen. Total functions let us stop some of those unexpected things happening, and so we can find any errors when the program is being compiled (called **compile-time**) rather than when it's being executed (called **run-time**).

The second definition is using the `(:)` value constructor operator as a pattern matcher. We know this value that we're pattern matching on must be a list with at least one item on it, because to get to the second definition, the first "empty list match" must have been passed over (we can say that it failed matching).

So, the `(:)` operator "pulls" the first item of the list out and "puts" it into the variable `x`, and then the "`_`" part throws the remainder of the list away, and this part of the function just returns the first item of the list as its result! This is pretty neat, isn't it?

So, does this mean we can pattern match with any function inside arguments like this? No. The `(:)` operator is actually part of the list type, and is used as we know, for constructing values. We've seen this happen when we write expressions like `1 : 2 : []`. We're using it there to build up a list. These functions are called **data constructors**, or **value constructors**. We can use them when in patterns for pattern matching, but we can't use ordinary functions, so don't expect to see `(+)` in a pattern match any time soon!

We know this will be a little hazy for you right now, but it will become clearer as we start to learn about more complicated data types later on. Don't worry too much about this for now.

9.14 Prefix Operator Pattern-Matching

You know that `(:)` is an operator, and operators can be changed from infix functions to prefix functions by using parentheses. Let's see what happens when we try to match using the `(:)` as a prefix function:

```
firstOrEmpty' :: [String] -> String
firstOrEmpty' []     = ""
firstOrEmpty' ((:) x _)  = x
```

Haskell has no problem with this at all, and nor should it. It turns out `((:) x _)` as a pattern means exactly the same thing as `(x:_)`. You probably won't see this very much in any source code you encounter, because it's less easy to read and write, but we're showing you here so that you understand that `(:)` is not syntax or special, it's just a data-constructor function, and they can all be used in patterns in the same way.

Let's see another function. This takes the first two elements of a `String` list and joins them together with a comma if there are at least two items in the list, otherwise it just returns the first one if there's one, otherwise an empty `String`.

```
firstOnesOrEmpty :: [String] -> String
firstOnesOrEmpty []       = ""
firstOnesOrEmpty [x]      = x
firstOnesOrEmpty (x:y:_)  = x ++ ", " ++ y
```

Fancy, right? You can use more than one `(:)` in your pattern matches. Notice in order to make sure our function was total, we had to add another definition. `[x]` matches only lists with exactly one item in them, and returns the single String inside. So you can use either the `(:)` value constructor, **or** the special `[]` syntax to pattern match variables out of lists. So we could rewrite the `firstOnesOrEmpty [x] = x` clause as `firstOnesOrEmpty (x:[]) = x` and it'd mean the exact same thing.

9.15 A Tiny Bit of Recursion

Now, we're going to blow your mind a little bit. Don't worry if this confuses you. Just read it carefully, and it might make sense. We'll cover this a lot more, so if it doesn't then that's fine.

What if we wanted something that would put comma-space between **all** of the elements of the list of strings and join them into one single string? Let's take a look:

```
joinedWithCommas :: [String] -> String
joinedWithCommas []     = ""
joinedWithCommas [x]    = x
joinedWithCommas (x:xs) = x ++ ", " ++ joinedWithCommas xs
```

Notice the last definition actually refers to itself! This is called **recursion**. We've stopped using "_" to "throw away" the rest of the list, we pattern-match it into a variable called `xs`, and then we use it... by passing it back into an application of the very same function we're defining!

What? Crazy! Completely cool, though! Recursion is an incredibly common and useful thing in Haskell.

This function is actually already implemented more generally in a Module called `Data.List` as the function named `intercalate`. We'll see it later when we use that module. It's more general because you can put **anything** between the items, not just `", "` as we have here.

9.16 The Final Shopping List Program

So our final program below prints out the count of items, and then the whole comma-separated list.

```
shoppingList :: [String]
shoppingList =
  [ "Carrots"
  , "Oats"
  , "Butter"
  , "Apples"
  , "Milk"
  , "Cereal"
  , "Chocolate"
  , "Bananas"
```

```
  , "Broccoli"
  ]

main :: IO ()
main = putStrLn ("There are "
                 ++ (show (length shoppingList))
                 ++ " items on the shopping list."
                 ++ " and the list is: "
                 ++ joinedWithCommas shoppingList)

joinedWithCommas :: [String] -> String
joinedWithCommas []     = ""
joinedWithCommas [x]    = x
joinedWithCommas (x:xs) = x ++ ", " ++ joinedWithCommas xs
```

9.17 Homework

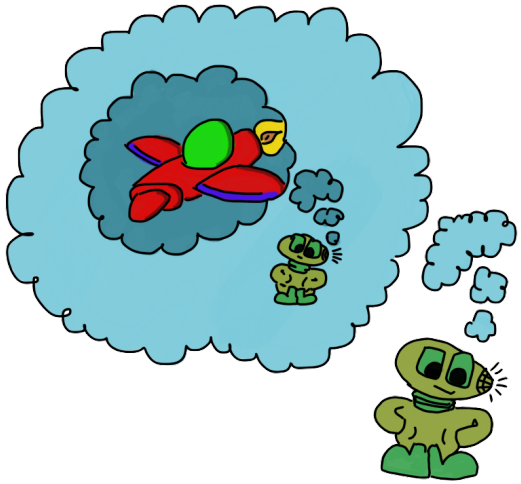
Your homework is to think about this recursive function, think it through, and also to run all the code.

Think about what `joinedWithCommas` would do if it was passed an empty list, and one `String`, then two `String`s, what would it do for three?

We'll explain it more soon, so if you don't understand it yet, it's to be expected and don't worry about it. This can take many months to understand properly.

10 A Dream Within a Dream

Let's say we have a movie collection, and for some reason, we're only interested in movies whose first letter is in the first half of the alphabet. In our program, we'll call those good movies, and the others bad.



```
movies =
  [ "Aeon Flux"
  , "The Black Cat"
  , "Superman"
  , "Stick It"
  , "The Matrix Revolutions"
  , "The Raven"
  , "Inception"
```

```
  , "Looper"
  , "Hoodwinked"
  , "Tell-Tale"
  ]
```

We'd like to be able to make a new list of our movies with "good" or "bad" appended to the name so we know which we can watch.

10.1 Predicates, String, Char

What we need is a function that decides if a movie is good:

```
isGood :: String -> Bool
isGood (x:_) = x <= 'M'
isGood _     = False
```

This kind of function, one that returns a Bool value that is depending on another value, is called a **predicate** function. It's yet another fancy word which comes to us from the subject of Logic. It simply means something can be either affirmed or denied about something else. That's exactly what's going on here. We're finding out if something is **good** or **bad**, by our arbitrary distinction, and returning a `Bool` value depending on that.

The first line takes any `String` of at least one element, matches x to the first element, then throws the rest away, and uses the `(<=) :: Ord a => a -> a -> Bool` operator to check if it's earlier in the alphabet than `'M'`.

There are some new things here, obviously. First is that we've got single quotation marks around M. What is the type of this thing? Well it's `'M' :: Char`. In Haskell, single quotation marks are used to indicate that something is a `Char` value.

As we mentioned before, a `String` is simply `[Char]`. That means `"Hello"` is the exact same thing as `['H', 'e', 'l', 'l', 'o']`. So, `String` has a special syntax, just like list has a special syntax. You can either write strings as a list of `Char` values, or you can write them with double quotation marks around them.

10.2 The Ord typeclass

Back to our predicate function, we notice we're using the `(<=)` function. This is called "less than or equal to". Its type indicates it takes two values of type constrained by the `Ord` typeclass. This typeclass is for types whose values can be compared in some ordered way (hence the name).

We're matching on the `String` argument using `(:)`, grabbing out the **head** as `x`, comparing it to the `Char` value `'M'` to see if it's in the first half of the alphabet, because `A` is "less than or equal to" `M`. The head of a list is the Haskell name for its first item, and the tail is the name for the remainder.

The `Ord` typeclass provides a type with the `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min` functions. They provide functionality to do various comparisons of values.

The end result of the above is a function that checks if a movie's title starts with a letter in the first half of the alphabet.

10.3 Transforming Titles

Now we'll see a function that takes a movie title and gives us a new name for it, depending on its good/bad assessment:

```
assess :: String -> String
assess movie = movie ++ " - " ++ assessment
  where assessment = if isGood movie
                     then "Good"
                     else "Bad"
```

In this function, we're introducing you to a `where` clause. This isn't an expression, so we can't embed `where` clauses anywhere we like, but rather they let us write definitions that are only "in effect" within the level above where they appear. This is called **local scoping**. That is, the **scope** of the definition only applies **locally** to the `where` clause.

This is why in the `assess` function above, we can use the definition for `assessment` within the function's body expression.

We can have multiple definitions and even functions written in `where` clauses. Let's see another definition of the assess function that has two definitions in its `where` clause:

```
assess' :: String -> String
assess' movie = movie ++ " - " ++ assessment
  where assessment = if movieIsGood
                     then "Good"
                     else "Bad"
        movieIsGood = isGood movie
```

Here we've just pulled `isGood movie` out into its own locally scoped definition. We don't recommend it, but if you wanted to get crazy, you could even put `where` clauses within definitions within other `where` clause definitions. Usually if you have this level of nesting, you should consider pulling your functions into smaller pieces.

10.4 Building Up a Function to Rename the Movie List

Next we're going to see a **partial function** so we can explain how it works to you in stages. Here, by **partial** we mean it's not **total**, because it's missing some cases of values of its arguments. That is, the function doesn't cover every possibility. This is **not a good thing to do**, ordinarily. We're just doing it here to show how we build up a function in steps.

So, this function we want to write should take a list of movies and give us back the newly named movie list...

```
assessMovies :: [String] -> [String]
assessMovies []  = []
assessMovies [y] = [assess y]
```

... but it doesn't, because it only works properly on lists of length 0 or 1, and we already know that in general, lists can be any length. The `[y]` pattern **only** matches against lists of one item, naming the item `y` as it does, and here we're returning a new list with the newly named movie in it by passing it to our `assess` function. We see we can have function applications inside of lists without trouble.

Next, we'll see that we add a pattern for lists of two items and change the whole thing to use the `(:)` operator as a pattern-matcher, because we know `[x]` and `(x:[])` are the same thing, but also to use the `(:)` operator to make the new list rather than how we did above:

```
assessMovies :: [String] -> [String]
assessMovies []       = []
assessMovies (y:[])   = assess y : []
assessMovies (x:y:[]) = assess x : assess y : []
```

This is a little better, but what about lists with more than two items? Also, it's getting tedious writing so many definitions, and a pattern of repetition is emerging. As we become programmers, we need to learn to see these repeated patterns, because they're usually an indication that something can be done in a better, or more abstract way.

One of the first patterns to notice is our second definition works for 1-item lists, and our third pattern-match has the exact same 1-item list pattern in it as the second definition.

If we could match the `(y:[])` part as one whole variable name in the third pattern, then somehow replace it with an application of the assessMovies function itself, we might be able to reduce these two definitions down to just one.

Well, you probably guessed that we **can** do this, and in fact this is exactly what we're going to do. This process of using the function we're defining within that function definition itself is called **recursion**, and it's one of the foundational underpinnings of Haskell.

So these two duplicated patterns can be resolved with a single definition, as long as we keep our empty list as a **base case**, otherwise we'd end up in an endless cycle. The empty list is our way out of the recursion. The recursive definition handles everything other than the empty list, which the base case handles.

```
assessMovies :: [String] -> [String]
assessMovies []     = []
assessMovies (x:xs) = assess x : assessMovies xs
```

So, if we had the list `["The Matrix"]`, that's equivalent to `"The Matrix" : []`, and so matches the second pattern with `"The Matrix"` as `x`, and `[]` as `xs`. The body of the function uses the list construction operator `(:)` to join the result of `assess x` to the application of `assessMovies` with its argument of the empty list, which results in the empty list as the base case, ending the recursion. You should think about how this works across a few different lists of different sizes.

Another way to think of this is if you had to crack some eggs. You could write down a definition for making cracked eggs like this:

"crack eggs" is: take a line of eggs. Crack the first egg, then do "crack eggs" on the rest of the eggs until there are none left. See how "crack eggs" is both the definition **and** used in itself? It's the same idea here, except Haskell creates a new list rather than modifies an existing one. We define building a list of assessed movies as: if the list is empty, give an empty list, otherwise take the first item and make its assessment String, joining that as the head of the list to the evaluation of the rest of the assessed movie titles list.

It's really beautiful, isn't it? You can start to see why having purity can really pay off. We can do this kind of substitution and equational reasoning easily. Why? precisely because we have pure functions, and because Haskell gives us these nice abstract pieces to work with.

This pattern - applying a function (the function `assess` here) to each element of a list — is so common that there's actually a function called `map` that extracts this mapping functionality. It takes a mapping function and a list as arguments. Its type is `map :: (a -> b) -> [a] -> [b]`. This type says that map is a function of two arguments, the first one of which is a function.

The `assess` function's type is `String -> String`. This fits into the pattern of the first argument of map: `a -> b`. In Haskell `a -> b` means the types **can** be different, but don't have to. It's a function of one type to a second type (which may or may not be the same type). So, `assess :: String -> String` fits the first argument to `map`. If we supply the function `map` with the function `assess` as its first argument, that means we're saying the type `a` in `map`'s type signature must be `String`, and also that the type `b` must be `String`, too. That means the second argument and the return type must both be `[String]`.

Let's rewrite `assessMovies` using `map` to get an intuition of how to use it, and build a small full program around it. If any of this is unclear, the next chapter will most likely clear it up as it delves more into functions such as `map`.

```
import qualified Data.List as L

movies =
  [ "Aeon Flux"
  , "The Black Cat"
  , "Superman"
  , "Stick It"
  , "The Matrix Revolutions"
  , "The Raven"
  , "Inception"
  , "Looper"
  , "Hoodwinked"
  , "Tell-Tale"
  ]

isGood :: String -> Bool
isGood (x:_) = x <= 'M'
isGood _     = False

assess :: String -> String
assess movie = movie ++ " - " ++ assessment
  where assessment = if isGood movie
                     then "Good"
                     else "Bad"

assessMovies :: [String] -> [String]
assessMovies = map assess
```

```
assessedMovies :: [String]
assessedMovies = assessMovies movies

main :: IO ()
main = putStrLn (L.intercalate "\n" assessedMovies)
```

First we import the `Data.List` package so we can use the intercalate function (which takes a joining list of a type, and a list of Lists the same type and builds a fresh list by joining each element of the list together with the joining item between). Here we're using it with `[Char]`, and `[[Char]]` which is the same as `String` and `[String]`, because `String` is just a type alias for `[Char]`.

If we were to define a "new" version of `String`, we could do it like this, because `String` is simply a type alias to a list of `Char`, otherwise known as a type synonym:

```
-- type sets up a type alias:
type NewString = [Char]
```

This is how we create a **type alias** (or **type synonym**). Once we have this in our program, everywhere Haskell sees `NewString` in your program, it will interpret it as `[Char]`. Note that these are not different types, they're identical, they just have different interchangeable names.

Anyway, after this, back in our main program, we set up the **movies** expression, and the `isGood` and `assess` functions. Then we create the `assessMovies` function which takes the list of movies and uses `map` with `assess` to build a list of assessed movies. Once that is done, we create the `assessMovies` expression that simply applies `assessMovies` to the movies list.

The `main` function then simply prints out with the passed in newline (`"\n"` is the special newline character in a `String`) between each of the `assessedMovies` (that's what `intercalate` does).

10.5 Homework

See if you can change the program so that it has a different first letter that decides which movies are bad and good.

After you've done that, see if you can change the program so that it has different movie titles in it.

11 More Shopping

Let's take a look at a program which will let us work out how much our shopping list will cost in total.

11.1 Tuples

To do this, we'll use a new type of data called a Tuple. A Tuple lets you keep some number of items of potentially different types together as one item. We can have 2-tuples, 3-tuples, and so on.

```
aShoppingListItem :: (String, Int)
aShoppingListItem = ("Bananas", 300)
```

This is a single shopping list item: a 2-tuple value. It has the `String "Bananas"`, and the `Int 300` which we're going to use to represent the number of cents that the bananas cost.

We can have tuples of different lengths. There are 3-tuples, and 4-tuples, and you can pretty much have as many as you'd like but it's best to just stick to two, or maybe three at the very most. There are better ways to build up composed data types that we'll see later on if you need to do that.

In the same way as we know that `[String]` is a type that can be expressed as `[] String`, we can express the 2-tuple `(String, Int)` as `(,) String Int`. In the same way, the 3-tuple `(Int, String, Int)` could be expressed as `(,,) Int String Int`, and so on. You can see the pattern. Note that each of the composed types can be any type at all. So, tuple types are created with the `(,)` style **type constructors**, there is actually an identically named **value constructor** for tuples. So you could just as easily write your tuple values as `("Bananas", 300)`, or as `(,) "Bananas" 300`.

11.2 Type Aliases (or Type Synonyms)

We actually want a list of these items, though. Let's take a look what it'd look like to have a new name for our `(String, Int)` tuple type so our program is more self-explanatory.

```
type ShoppingListItem = (String, Int)

aShoppingListItem :: ShoppingListItem
aShoppingListItem = ("Bananas", 300)
```

We use "`"type"`" to tell Haskell we're defining a **type alias** (or **type synonym**). This means Haskell sees `ShippingListItem` as being the same type as `(String, Int)`. This is just to make our programs more readable for us. Haskell won't see these types as different, so if you accidentally used a `(String, Int)` where you meant to use a `ShoppingListItem`, then Haskell won't complain. Ideally, we'd like it to, though.

Note that in Haskell, all types must start with a capital letter, and all variable names must start with a lowercase letter.

Ok so let's look at another couple of type synonyms to make things clearer, and also a type synonym for shopping lists, and an actual shopping list, too:

```
type Name = String
type PriceInCents = Int
type ShoppingListItem = (Name, PriceInCents)
type ShoppingList = [ShoppingListItem]

shoppingList :: ShoppingList
shoppingList = [ ("Bananas", 300)
               , ("Chocolate", 250)
               , ("Milk", 300)
               , ("Apples", 450)
               ]
```

So `Name` is now `String`, `PriceInCents` is `Int`, `ShoppingListItem` is a tuple of `Name` and `PriceInCents`, which is the same as saying it's a tuple of `String` and `Int`, and `ShoppingList` is the same thing as `[(Name,PriceInCents)]`, which is a list of tuples, and is the same thing as `[(String,Int)]`.

All of these type aliases makes it much easier to understand what the programmer who wrote this intended. Especially the type `PriceInCents`. If we didn't have this, we would have no idea what the numbers in each tuple are supposed to represent. We would have to either work it out by looking at all of the code, or hope that the programmer had written some helpful comments in the code.

11.3 The Final Program

Let's look at the finished program that will tell us how much the total price of a shopping list is, in cents:

```
type Name = String
type PriceInCents = Int
type ShoppingListItem = (Name, PriceInCents)
type ShoppingList = [ShoppingListItem]

shoppingList :: ShoppingList
shoppingList = [ ("Bananas", 300)
               , ("Chocolate", 250)
               , ("Milk", 300)
               , ("Apples", 450)
               ]

sumShoppingList :: ShoppingList -> PriceInCents
sumShoppingList []     = 0
sumShoppingList (x:xs) = getPriceFromItem x +
                         sumShoppingList xs

getPriceFromItem :: ShoppingListItem -> PriceInCents
getPriceFromItem (_, price) = price

main :: IO ()
main = putStrLn ("Price of shopping list is "
                 ++ show (sumShoppingList shoppingList)
                 ++ " cents.")
```

We have two new functions to look at here.

The first is `getPriceFromItem`. The name pretty much explains exactly what it does. It uses pattern matching on a `Shop-`

`pingListItem` (which is a tuple), and extracts only the second element of the tuple.

There are actually two functions that work with tuples called `fst` and `snd` that pull out the first or second element of a tuple respectively. We could have just defined `getPriceFromItem` as being `snd`, because we've pretty much just re-created it here, but it's useful to show you how to do it.

11.4 More Recursion Explained

The second new function is `sumShoppingList`. This is using recursion to go over each item in the list, and apply the `getPriceFromItem` function to them, adding the prices together as it does so.

When `sumShoppingList` is evaluated, one way to think about how it can do the work to find a value is to imagine what it would look like if all of the expansions of `sumShoppingList` had already taken place inside the body of the function. This is, then, an equivalent expression to `sumShoppingList shoppingList`:

```
getPriceFromItem ("Bananas", 300)
+ (getPriceFromItem ("Chocolate", 250)
  + (getPriceFromItem ("Milk", 300)
    + (getPriceFromItem ("Apples", 300)
      + 0)))
```

After this we can imagine what it'd be like after all the `getPriceFromItem` functions have been applied to their tuple arguments, this is an equivalent expression:

```
300 + 250 + 300 + 450 + 0
```

It's pretty easy to see how this is equal to 1300.

11.5 Folding

Going from the expanded recursion form to the single number is an example of what's called **folding**, and it involves reducing a list to a single value.

We've seen this pattern a fair bit so far. Let's look at it some more with a few more recursive functions:

```
joinStrings :: [String] -> String
joinString  []     = ""
joinStrings (x:xs) = x ++ joinStrings xs

sumIntegers :: [Integer] -> Integer
sumIntegers []     = 0
sumIntegers (x:xs) = x + sumIntegers xs

-- subtracts all subsequent numbers from
-- the first numbers
subtractNums :: Num a => [a] -> a
subtractNums []     = 0
subtractNums (x:xs) = x - subtractNums xs
```

```
productOfIntegers :: [Integer] -> Integer
productOfIntegers []     = 1
productOfIntegers (x:xs) = x * productOfIntegers xs
```

If you look across all of those functions, and try to see what's similar about them, you may notice some interesting things:

Firstly, there is a single value for the empty list case. This is called the **base value**. The case is called the **base case**, because it's where the recursion ends.

Secondly, there is an operation being applied between each element of the list. For joinStrings, it's `(++)`. For `productOfIntegers`, it's `(*)`. This is called the **folding function**, because it's what Haskell uses to do the folding after the recursion has been fully expanded.

Thirdly, and a little more subtle, is that all of these functions fold **to the right**, which means the recursive function application happens at the right. This is why they're called **right folds**. If we had our recursion on the left, it works differently.

11.6 Using foldr

If you remember, functions are values as well, which means we can pass a function into another function as a value. In fact, if you rememeber, **all** functions of more than one argument do this. Let's look at all the examples of the above, rewritten using the generalised fold right function:

```
joinStrings :: [String] -> String
joinStrings xs = foldr (++) "" xs

sumIntegers :: [Integer] -> Integer
sumIntegers xs = foldr (+) 0 xs

subtractNums :: Num a => [a] -> a
subtractNums xs = foldr (-) 0 xs

productOfIntegers :: [Integer] -> Integer
productOfIntegers xs = foldr (*) 1 xs
```

So, it seems `foldr` takes **three** arguments! The first is a function of two arguments: the folding function. The second is the base value, and the third is the list we're folding over.

This is what the type signature of `foldr` looks like:

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Just like `length` is generalised to work on any kind of `Foldable`, so is `foldr`. Actually, `foldr` and `length` are part of the `Foldable` typeclass.

Let's imagine what the type of `foldr` would actually be specialised to for the `joinStrings` function, given that it's dealing only with lists as its `Foldable` type.

First, the `Foldable` instance we're dealing with is the one for List, so we could replace the "`Foldable t => t`" part with List like this:

```
foldr :: (a -> b -> b) -> b -> [] a -> b
```

but we can just write `[] a` as `[a]`, so:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Okay, the `Foldable => t` constraint and type is now replaced with List, because `Foldable` related to the container type, which is List, and that is now fully specified.

Next, what about the type of `a`? Well, we know `foldr` takes three arguments, and the third is a list of `String` in the body of `joinStrings` because it looks like this: `joinStrings xs = foldr (++) "" xs` and `xs` is of type `[String]`.

So, `a` must be `String`. We take a List of Strings. While we're at it, b will be String, too (because the `(++)` function, which we're using, has the same type on both of its arguments. So we can rewrite the type for `foldr` as specialised inside `joinStrings` like this:

```
foldr :: (String -> String -> String)
         -> String
         -> [String]
         -> String
```

That makes more sense in that case. We fold over our list of strings, joining them together with `(++)` as we go, and our base case for when we have only got the empty list left is the empty string `""`.

Next, though, we'll look at `sumShoppingList` again, and how to make it use `foldr`. We'd just like to remind you about the type of the folding function in `foldr`. It's `(a -> b -> b)`, and that means that first argument **can** be a different type than the second, but doesn't have to, but that the second one **must** be the same type as the result.

```
sumShoppingList :: ShoppingList -> PriceInCents
sumShoppingList []     = 0
sumShoppingList (x:xs) =
  getPriceFromItem x + sumShoppingList xs

sumShoppingList' :: ShoppingList -> PriceInCents
sumShoppingList' xs = foldr getPriceAndAdd 0 xs

getPriceAndAdd :: ShoppingListItem ->
                  PriceInCents ->
                  PriceInCents
getPriceAndAdd item currentTotal =
  getPriceFromItem item + currentTotal

getPriceFromItem :: ShoppingListItem -> PriceInCents
getPriceFromItem (_, price) = price

main :: IO ()
main = putStrLn ("Price of shopping list is "
                 ++ show (sumShoppingList shoppingList)
                 ++ " cents.")
```

So, `sumShoppingList'` is our function that uses `foldr`, which also uses another new function `getPriceAndAdd`, whose type could be thought of as `a -> b -> b` which matches the first argument of `foldr`, and `0` is our base value just like in our recursive

version.

11.7 Built in Recursive Functions

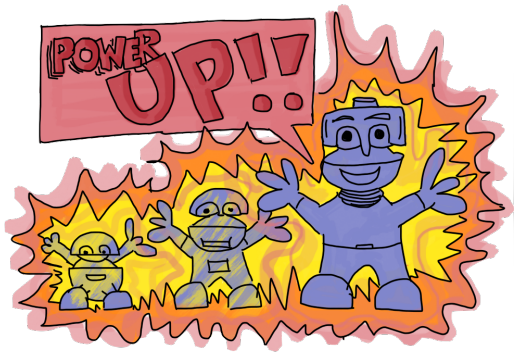
Haskell already has "built-in" functions for `sum`, `product` and joining strings together (`concat` generally concatenates lists of the same type together, so will work fine on `String` values). Here, we've just been illustrating how they could be built to show you how basic recursion works. The built-in functions are more generalised to work on any `Foldable` instance.

11.8 Homework

Your homework is to go through all the functions we saw, think about how they work, using pen & paper to work out the steps involved in evaluating them applied to some values of your own choosing.

12 How To Write Programs

At this point, we've seen enough Haskell in actual programs that we can start to level up our learning to the next stage: adjusting and writing extremely simple programs.



We haven't yet seen all of Haskell's basics even once through yet, though, so we continue our tour of reading basic programs as we do.

A warning at this point: as Han-Solo says to Luke in Star Wars: "Don't get cocky, kid!"

That is, don't try to do too much at this point and don't expect too much from yourself. Just proceed little by little. Build up piece by piece. Before you know it, you'll be at a point where you can write small but enjoyable programs, and you'll know how to begin to find things for yourself when you don't know!

In other books, writing programs is often left as an exercise for the reader to work out how to do, but this doesn't sit well with us, and it usually doesn't work very well, either.

We prefer to guide you, because the task of problem solving is not necessarily obvious. In order to write a program, one strategy is to pull the problem its trying to solve apart into smaller pieces. This is called **step-wise refinement**, or **top-down design**.

We can also go the other way: (**Bottom-up design**, obviously!) start with the things we already know we can do. Smaller, known pieces, then build up with those to form larger and larger pieces by combining them until we have our complete program.

Professional programmers use both of these strategies all the time to piece programs together and solve problems.

Haskell is very good at helping us to do both of these things. Its type system shows us where things match up, and its functional nature lets us easily separate data from code which keeps things free to be reused, re-combined and composed.

We use processes called **equational reasoning**, **function extraction**, **dummying**, **black-boxing** (otherwise known as **encapsulation**) and **refactoring** to do these strategies; identical processes that software professionals use every day to get their work done, sometimes without realising it.

Let's continue.

13 At The Zoo

In this chapter, we'll make a small program that tells a Zoo owner what advice to take for each animal in the Zoo if it escapes. In the process, we'll introduce you to an awesome feature of Haskell: the ability to make your own data types, and values of those types.

We'll have a value for each of a number of animals, and we'll also have a list of animals which we'll call a Zoo.

To do this, we'll use the `data` keyword which creates a new data type, and specifies all the values it can possibly be. Another name for these kinds of types is a **sum type**, because the values of the type "summed together" make up the whole type.

13.1 Sum Types

So, let's see a sum type.

```
data Animal = Giraffe
            | Elephant
            | Tiger
            | Flea
```

We're saying that we want Haskell to make a new type, named `Animal`. We're also saying that the data for the `Animal` type can be any one of `Giraffe`, `Elephant`, `Tiger` or `Flea`, but nothing else. These values are also called **value constructors**, even though they're each only able to construct the value that they are. We'll see why more later.

Let's see the type for `Zoo` next:

```
type Zoo = [Animal]
```

Pretty simple. A `Zoo` is an `Animal` list. You might recognise this is just a type synonym, for our convenience and documentation. Next we'll see a definition for a `Zoo`:

```
localZoo :: Zoo
localZoo = [ Elephant
           , Tiger
           , Tiger
           , Giraffe
           , Elephant
           ]
```

Ok, the `Zoo` named `localZoo` has some `Animal` values in it. Let's put all of this together and add a function that uses a `case` expression to give some advice when a particular animal escapes.

13.2 Pattern Matching with Sum Types

Below, we have a function that takes a single `Animal`, and returns a piece of advice as a `String`, for when that `Animal` escapes. Looking at the `case` expression, you can see it's matching against the values of the `Animal` data type.

```
data Animal = Giraffe
            | Elephant
            | Tiger
            | Flea

type Zoo = [Animal]

localZoo :: Zoo
localZoo = [ Elephant
           , Tiger
           , Tiger
           , Giraffe
           , Elephant
           ]

adviceOnEscape :: Animal -> String
adviceOnEscape animal =
```

```
case animal of
  Giraffe   -> "Look up"
  Elephant  -> "Ear to the ground"
  Tiger     -> "Check the morgues"
  Flea      -> "Don't worry"
```

If you remember how `case` expressions work, the `animal` variable is checked against each of the left hand side patterns to see if it maches, and if it does, the right hand side expression (here a `String` value) will be returned.

Do you notice that there's no **default** case, usually marked with an underscore? That's because we **know** this function is total already, because it has one item for each of the possible data values of the `Animal` type, so there's nothing left to catch for a default case.

13.3 More Recursion

Next we're going to look at a function that takes a `Zoo`, and returns a list of all the advice for when all the animals in that `Zoo` escape, by using the `adviceOnEscape` function and recursion.

```
adviceOnZooEscape :: Zoo -> [String]
adviceOnZooEscape []     = []
adviceOnZooEscape (x:xs)  =
  adviceOnEscape x : adviceOnZooEscape xs
```

Maybe you recognise this code as recursion, and similar to the previous chapters, in that it has a kind of a "folding" shape.

It's a little bit different, though, because we can't just use `(:)` as the folding function, as we're also applying `adviceOnEscape` to each item as we fold them together into the new list.

In fact, in this case while we **could** think of it as folding the list into another list, we're not really folding the list down to a single value, we're just applying a function across all the elements of the list. Another way to look at it is that we're making a new list that is just like the old one with a function applied to all its elements.

We could try a fold to do this, but we'd have to extract both the `adviceOnEscape` and the `(:)` out into a single folding function. Let's see what that would look like, and we'll also see some more local binding using a `where` clause while we do so:

```
adviceOnZooEscape :: Zoo -> [String]
adviceOnZooEscape []     = []
adviceOnZooEscape (x:xs)  =
  adviceOnEscape x : adviceOnZooEscape xs

adviceOnZooEscape' :: Zoo -> [String]
adviceOnZooEscape' xs =
    foldr addAdviceForAnimal [] xs
  where addAdviceForAnimal animal adviceList =
          adviceOnEscape animal : adviceList
```

When we look at them together, we can see that we're worse off that before! `foldr` was supposed to help us write less code, but it actually has us producing more! This should be telling us something: that `foldr` is not the right abstraction to use here.

We included a function called `addAdviceForAnimal`, which we used as our folding function. As we've seen before, the `where` clause handily makes definitions below it available to the `adviceOnZooEscape'` function. This is called **local scoping**. The `where` clause makes the definitions witin it only available within the expressions that appear above it. We'll see more of this soon.

So, it turns out that there is actually a function whose job it is to do what we want here: take a list of `a` and turn it into a list of `b`, using a function of type `(a -> b)` (which we could call the **mapping function**). It's called `map :: (a -> b) -> [a] -> [b]`, because it keeps the "shape" of the list, but through across all its values and applies the mapping function to each item, producing a new list of mapped values as it does. Let's see it in action:

```
adviceOnZooEscape :: Zoo -> [String]
adviceOnZooEscape xs = map adviceOnEscape xs
```

Really nice, and clean looking. The map function is called a **higher order function** because it takes a function as an argument, so it's an order higher than normal functions that just take values.

Also, we can simplify this by not mentioning the argument to the function. It still has one, but it's implied by the types of the function and `map`. Observe:

```
adviceOnZooEscape :: Zoo -> [String]
adviceOnZooEscape = map adviceOnEscape
```

You give a 2-argument function only 1 argument, and this turns it into a 1-argument function! Haskell rocks!

We're defining `adviceOnZooEscape` as a function of one argument without mentioning the argument it takes. We can do this because we're also not mentioning the second argument of `map`. Another way to say this is that we've made an expression of `map`, a 2-argument function, and we've only given it one of its arguments, so the result is a function of one argument.

13.4 What is Currying?

You may remember that a function `plus :: Int -> Int -> Int` can be defined as `plus x y = x + y`, or it can be defined as `plus = \x -> (\y -> x + y)`, they're identical to Haskell, because a 2-argument function is actually a function that returns a function of one argument. If we give `plus` one `Int` value, it will bind that value to `x`, and return the inner function. Let's see a defition for `plus5`: `plus5 = plus 5`. This will return the function `y -> 5 + y`. This way of defining multiple argument functions is called **currying**. It's named after one of the men who invented it, **Haskell Curry**. Yes, Haskell is named after him.

13.5 The Finished Program

Next we'll see how this connects up to a comma-separating function, and a definition for main to finish the program, and we'll use another

where clause:

```
import qualified Data.List as L

data Animal = Giraffe
            | Elephant
            | Tiger
            | Flea
type Zoo = [Animal]

localZoo :: Zoo
localZoo = [ Elephant
           , Tiger
           , Tiger
           , Giraffe
           , Elephant
           ]

adviceOnEscape :: Animal -> String
adviceOnEscape animal =
  case animal of
    Giraffe   -> "Look up"
    Elephant  -> "Ear to the ground"
    Tiger     -> "Check the morgues"
    Flea      -> "Don't worry"

adviceOnZooEscape :: Zoo -> [String]
adviceOnZooEscape = map adviceOnEscape

joinedWithCommasBetween :: [String] -> String
joinedWithCommasBetween []     = ""
joinedWithCommasBetween [x]    = x
joinedWithCommasBetween (x:xs) =
  x ++ ", " ++ joinedWithCommasBetween xs
```

We have some new things here. Firstly, we have a **qualified import**. Importing is how we can include other code to use in ours. Making it qualified means all the imports actually sit underneath a special name (we're calling it `L` here), and so as you can see, when we want to use the `intercalate` function below, in main's definition, we have to write `L.intercalate` to tell it we mean the one inside the `Data.List` module.

The second thing to note is that we've included a `joinedWithCommasBetween` function. We're not actually using it here. We've seen it before, but it's identical to the function obtained by providing the `intercalate` function from `Data.List` with a `", "` value, except that it also works on any Lists, not just `[String]`, so we included the definition so you can understand one way `intercalate` could work.

The type of `intercalate` is `[a] -> [[a]] -> [a]`. Because the `String` type is actually just a synonym for `[Char]`, this fits if "`a`" is `Char`. (It fits as `String -> [String] -> String`).

The type of the expression `intercalate ", "` is `[String] -> String`, same as `joinedWithCommasBetween`.

We're using two lines in a where clause inside our `main`, this time. There is `advices`, which uses `adviceOnZooEscape` to build a list of pieces of advice using `localZoo`, and `stringToPrint` which uses `intercalate` and `advices` to create a string that is passed to `putStrLn`.

13.6 Homework

Your homework is to write a program that prints out the `String` `"Hello there"`, and then change it to print out your name. Try to remember what you have to write and not look at the book while you're writing your program. Only once you've finished, check your work against the book, and by running the program.

Once you've done that, do it again, but make the program print out your mother's name.

Once you've done that, do it again this time with your favourite colour.

Do this by both using a separate definition for the `String`, as well as putting the `String` directly in.

You should do this as many times as you need to with different `String` values, not looking until after, so that you can write any program that prints out a `String` without looking anything up. Make sure you're writing the type signatures, as well.

Now it's time to pat yourself on the back, and take a break - you've written your first Haskell programs! Well done.

At this point, you know how simple function application works. You take a value, and you put it to the right of a function, and this expression in total is equal to the returned value.

Why did we wait so long before recommending you begin to write code? Simply, we want you to be **very** comfortable with seeing, reading and understanding things you write before you begin to write them.

14 Cats and Houses

Feline Crescent has ten houses on it. Each house has a cat of a different breed living in it. We'll be seeing sections of programs that we'll slowly build up to a program that finds the oldest of the cats on the street and prints out where they live, along with their age in equivalent human years.

14.1 Another Sum Type

First, let's see a data type for cat breeds:

```
data CatBreed =
    Siamese | Persian | Bengal | Sphynx
  | Burmese | Birman | RussianBlue
  | NorwegianForest | CornishRex | MaineCoon
```

If you remember, we know this **sum type** means we're declaring a new type of data called `CatBreed`, and all the possible values are listed above.

14.2 Product Types and Algebraic Data Types

Next, we'll see a data type for `Cat`. This is a new kind of data type called a **product type**. This lets us make values that combine more than one type. When we saw **tuples**, you may rememeber we said that there are better ways to combine multiple values into one value. Well, this is that better way. A `Cat` can have an `Age`, and a `Name`, and a breed (`CatBreed`).

```
type Name = String
type Age = Integer
data Cat = Cat Name CatBreed Age
```

This tells Haskell that `Cat` is a new type for data, and that it has only one value constructor. Both **sum types** and **product types** are

examples of **algebraic data types**. You can have combinations of these types in the one type, which means you can have types that are sums of product types. We'll see more of this later, so don't worry too much about this for now. Algebra is another fancy word which just means a sort of language of combining elements together — in this case, we're combining types. You probably know this word from Maths if you've studied it.

Our code also tells Haskell that `Cat` is the name of the value constructor, as well as the name of the type. So how does Haskell know when we're talking about the type, and when we're talking about the value? Well, by looking at the places we're using it, it can use inference to work it out.

Notice the `Cat` type is defined as one single `Cat` data constructor, which has variable "slots" for the name, breed and age. This actually tells Haskell to make the value constructor function `Cat :: Name -> CatBreed -> Age -> Cat` and this can be used to make `Cat` values (which is why it's called a data constructor), as well as pattern match for these values. Remember how `(:)` is a value constructor for the list type? Well, it's the same thing at work here.

14.3 Pattern-Matching Product Types

Next we'll see a product type for House:

```
type HouseNumber = Int
data House = House HouseNumber Cat
```

And a function to work out how old a cat is in human years:

```
-- this is a commonly agreed upon
-- way to work out cat ages
humanAge :: Cat -> Age
humanAge (Cat _ _ catAge)
  | catAge <= 0 = 0
  | catAge == 1 = 15
  | catAge == 2 = 25
  | otherwise   = 25 + (catAge - 2) * 4
```

We're using a guard pattern to match all the possibilities here. Maybe you can remember that the `(<=)` operator means "is less than or equal to". There is a similar operator for the other direction, too `(>=)` which means "is greater than or equal to".

Notice that the first argument to humanAge is a `Cat`, a single value of the `Cat` type, but it's being kind of "pulled apart" by the pattern match using the value constructor. This takes the first two **fields** of the `Cat` data type and ignores them (by matching them to "`_`" which as you might know by now, basically throws them away), and then binds a variable name to the age of the `Cat` called `catAge`, so the rest of the function can compare and do math with it.

Next we'll see some data for a `street`, which will be a list of houses, and a couple of functions for working with that data:

```
street :: [House]
```

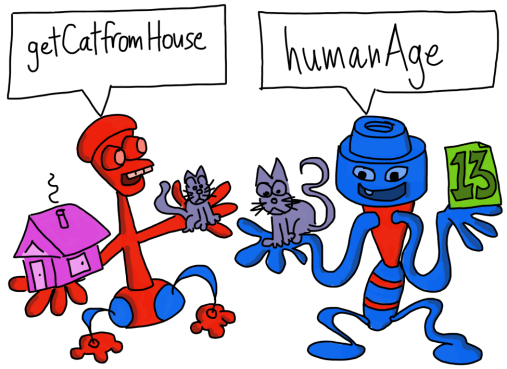
```
street =
  [ House 1 (Cat "George" Siamese 10)
  , House 2 (Cat "Mr Bigglesworth" Persian 5)
  , House 3 (Cat "Mr Tinkles" Birman 1)
  , House 4 (Cat "Puddy" Burmese 3)
  , House 5 (Cat "Tiger" Bengal 7)
  , House 6 (Cat "The Ninja" RussianBlue 12)
  , House 7 (Cat "Mr Tinklestein"
                 NorwegianForest
                 8)
  , House 8 (Cat "Plain Cat" MaineCoon 9)
  , House 9 (Cat "Shnooby" Sphynx 7)
  , House 10 (Cat "Crazy Ears Sam"
                  CornishRex
                  3)
  ]

getCatFromHouse :: House -> Cat
getCatFromHouse (House _ c) = c

getHumanAgeOfCatFromHouse :: House -> Age
getHumanAgeOfCatFromHouse =
  humanAge . getCatFromHouse
```

So, `street` is a value whose type is `[House]`. The type of `House` says it has a single data constructor, also called `House` which takes two fields (`House :: HouseNumber -> Cat -> House`) and returns a `House` value. Each of these houses has an embedded `Cat` value in it constructed with the `Cat` data constructor which is building a `Cat` out of a `Name`, its `CatBreed`, and its `Age`.

Moving on from there, we know that every `House` must have a `Cat` in it if we look at the `House` type, so we see there's a simple function that extracts the `Cat` value from a `House` value by using pattern matching (called `getCatFromHouse`).

14.4 Function Composition

The next function (`getHumanAgeOfCatFromHouse`) probably looks a little curious, because we've got a new operator in it named `(.)` which kind of glues two functions together. It turns them into a single function that does the same as calling the first one on the result of calling the second one. It's called the **function composition operator**. We'll talk more about this function later, but you can think of it as feeding the "output" of the function on the right into the "input" of the function on the left.



The type of `getHumanAgeOfCatFromHouse` is annotated to be `House -> Age`. As we just saw, `getCatFromHouse` takes a

`House` and gives a `Cat`, and `humanAge` takes a `Cat` and gives an `Age`. So, if we somehow could chain or pipe them together (glue them at the `Cat`, so to speak!), then giving this new function a `House` value would give us back an `Age`. This is exactly what the `(.)` operator does to two functions: it chains them together to form a new one that does the work of both, as long as they have a common type between them.

Here's another way to write that same relation by just using regular function application rather than function composition.

```
getHumanAgeOfCatFromHouse :: House -> Age
getHumanAgeOfCatFromHouse h =
  humanAge (getCatFromHouse h)
```

We can see that we have to include a variable for the `House` value here, and we can see that we're applying the function `getCatFromHouse` to the `House`, and then applying the function `humanAge` to the resultant `Cat`.

Sometimes it makes more sense to use normal function application, like the above: `humanAge (getCatFromHouse h)` and other times it makes more sense to use function composition like this: `humanAge . getCatFromHouse`, but they mean the same thing. We'll see more of `(.)` later.

14.5 Importing a Module

Now we'll see a function from the `Data.List` module called `find` that can be used to get a particular item from a List. A module is a kind of named package of additional functions and types that other programmers have written. So, the `Data.List` module contains lots of good things to do with lists, and it needs to be imported in our file at the top. We're "aliasing" it (or locally renaming it) to `L` by using the `as` keyword. The `qualified` keyword makes sure when it imports all the function names, it doesn't load them directly into our local **namespace**. That way, if we already have things named the same thing as the module we're importing, there won't be any conflicts.

```
-- don't forget, this goes
-- at the top of the file
import qualified Data.List as L
```

14.6 Maybe An Answer

Let's look at the type signature for the `find` function, noticing that we've **qualified** the function as being in the `L` aliased namespace by putting `L.find` instead of just `find`:

```
import qualified Data.List as L

L.find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

We can see this function takes two arguments; firstly a function of type `a -> Bool`, and secondly a value of type `Foldable t => t a`. This function then returns another wrapper type. This one is called `Maybe`, and it is wrapping the same type that our `Foldable t => t` is wrappering.

What does this function do, though? Well, it takes that function of `a -> Bool` and applies it to each of the items in the `Foldable t => t a`. If it can't find any that return `True`, it returns the `Nothing` value, which comes from the `Maybe a` type. If it **does** find any that return `True`, it returns the first one, wrapped in the `Just` value constructor from the `Maybe a` type.

Here's an example:

```
names = ["Harry", "Larry", "Barry"]

result1 = L.find isHarry names
  where isHarry name = name == "Harry"
-- result1 will be:
-- Just "Harry"

result2 = L.find isJake names
  where isJake name = name == "Jake"
-- result2 will be:
-- Nothing
```

So we can easily see when it finds a value because `True` was returned by the finding function (also called the **predicate**, remember?), it returns it wrapped it in `Just`, and when it doesn't, it returns `Nothing`.

14.7 A Little Finding & Sorting

Let's put this to work on a function that can find the oldest cat in a list of houses:

```
findOldestCat :: [House] -> Maybe Cat
findOldestCat []     = Nothing
findOldestCat houses = Just oldestCat
  where
    oldestCat
      = getCatFromHouse houseWithOldestCat
    houseWithOldestCat
      = head housesSortedByCatAge
    housesSortedByCatAge
      = L.sortBy catAgeComparer houses
    catAgeComparer (House _ (Cat _ _ age1))
                   (House _ (Cat _ _ age2))
      = compare age2 age1
```

This function might look insane at first, but it's just layers on layers, and we're going to slowly pull the layers apart together. It'll be fun, let's start.

Let's go from the bottom up. `catAgeComparer` is a function that takes two houses and compares the ages of the cats contained within. It does this by pattern matching the ages out of the cats, and the cats out of each house all at once (its type is `House -> House -> Ordering`).

An `Ordering` is a built-in **sum** data type which has values of `LT`, `EQ` and `GT` which stand for less than, equal to and greater than

respectively. `Ordering` values are used by sorting functions in Haskell.

The `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` function from `Data.List` takes a function whose type is `(a -> a -> Ordering)` to sort its second argument: a list. That fits the type of `catAgeComparer`, which is why we're using `sortBy` in our definition of `housesSortedByCatAge` in the line above that. Put another way, the `sortBy` function takes a comparing function (that is, one that returns an `Ordering`), and a list, and returns that list sorted by using the comparing function on adjacent elements.

Because we want to sort oldest to youngest, our application of the compare function in `catAgeComparer` has `age2` first. If `age1` was first, it'd sort youngest to oldest.

Next, we have `houseWithOldestCat` which takes the `housesSortedByCatAge` value obtained by sorting the houses, and picks off the first one with the `head` function. It's only safe to use the `head` function when we can be absolutely sure there is at least one item in the list we're applying it to. Otherwise it will cause your program to crash (crashing is a term that means the program unexpectedly stopped working). We're sure that there is at least one item in the list we're applying `head` to because we have a clause that matches on the empty list and returns `Nothing`.

Finally, `oldestCat` is obtained by applying `getCatFromHouse` to `houseWithOldestCat`, which will obviously just get the cat out of the house.

14.8 More About Maybe

Now we can talk some more about the `Maybe Cat` type that you can see in the `findOldestCat` function's type signature.

Maybe is a type that takes another type to make types with (this is called a **type constructor**). That means you can have values of type `Maybe Int`, `Maybe Cat`, `Maybe House`, or `Maybe` any other concrete type you like.

It's a sort of wrapper for other types, and it's what we use in Haskell when we want to make a type's value optional.

In our case, we can't be 100% sure if the list we pass to `findOldestCat` will contain something. If it is empty, then we obviously can't pass a `Cat` back at all. The way `Maybe` values work is there's a value called `Nothing` which represents "no value" of the wrapped type, and there's a value called "`Just a`" which represents any other value of our wrapped type. Take a look at the following values and their type signatures:

```
Just 5 :: Num a => Maybe a
Just "Heya" :: Maybe String
Just (Cat "YOOBEY" Sphynx 8) :: Maybe Cat
Nothing :: Maybe Cat
Nothing :: Maybe Integer
Nothing :: Maybe a
```

You might be surprised to know that `Nothing :: Maybe Cat` can't be compared to `Nothing :: Maybe Integer`. This

is because `Cat` is a different type than `Integer` and you can't compare differently typed values (unless they're **polymorphic** values — that is, values whose types have type variables like `Nothing :: Maybe a`, or `5 :: Num a => a`).

So, `Nothing :: Maybe Cat` means "there are no cats", and `Just (Cat "YOOBEY" Sphynx 8)` means "a value of the optional-`Cat` type that has a `Cat` in it). This is different than the values whose type is `Cat`, because a value of the type `Cat` **must** be a `Cat` as defined by the type - it can't be empty at all.

This is a very nice property for a programming language to have. If there is a value whose type is `Integer`, you can be sure it won't have anything other than exactly that in it, which makes reading and reasoning about programs much much easier.

However, every positive side has a negative side, too, and the negative side of this is that it makes working with empty things slightly more complicated than if there were just a general value that means an empty thing. We definitely think the complexity is worth it, though, because these **optional types** are some of the biggest sources of errors in programming languages that don't have this feature of "typed optionality".

14.9 The Final Program

We've added a `main` function and two additional helper functions:

```haskell
import qualified Data.List as L

data Cat = Cat Name CatBreed Age
type Name = String
data CatBreed =
    Siamese | Persian | Bengal | Sphynx
  | Burmese | Birman | RussianBlue
  | NorwegianForest | CornishRex | MaineCoon
type Age = Integer

data House = House HouseNumber Cat
type HouseNumber = Int

street :: [House]
street =
  [ House 1 (Cat "George" Siamese 10)
  , House 2 (Cat "Mr Bigglesworth" Persian 5)
  , House 3 (Cat "Mr Tinkles" Birman 1)
  , House 4 (Cat "Puddy" Burmese 3)
  , House 5 (Cat "Tiger" Bengal 7)
  , House 6 (Cat "The Ninja" RussianBlue 12)
  , House 7 (Cat "Mr Tinklestein"
                 NorwegianForest
                 8)
  , House 8 (Cat "Plain Cat" MaineCoon 9)
  , House 9 (Cat "Shnooby" Sphynx 7)
  , House 10 (Cat "Crazy Ears Sam"
                  CornishRex
                  3)
  ]

humanAge :: Cat -> Age
humanAge (Cat _ _ catAge)
  | catAge <= 0 = 0
  | catAge == 1 = 15
  | catAge == 2 = 25
```

```
  | otherwise   = 25 + (catAge - 2) * 4

getCatFromHouse :: House -> Cat
getCatFromHouse (House _ c) = c

getHumanAgeOfCatFromHouse :: House -> Age
getHumanAgeOfCatFromHouse =
  humanAge . getCatFromHouse

findOldestCat :: [House] -> Maybe Cat
findOldestCat []     = Nothing
findOldestCat houses = maybeOldestCat
  where
    maybeOldestCat
      = case findOldestCatHouse houses of
          Just house ->
            Just (getCatFromHouse house)
          Nothing ->
            Nothing


findOldestCatHouse :: [House] -> Maybe House
findOldestCatHouse houses =
    if length housesSortedByCatAge > 0
    then Just (head housesSortedByCatAge)
    else Nothing
  where housesSortedByCatAge
          = L.sortBy catAgeComparer houses
        catAgeComparer (House _ (Cat _ _ age1))
                       (House _ (Cat _ _ age2))
          = compare age2 age1

getCatName :: Cat -> String
getCatName (Cat name _ _) = name

getHouseNumber :: House -> String
getHouseNumber (House number _) = show number

main :: IO ()
main = putStrLn oldest
  where
    oldest =
      case findOldestCatHouse street of
        Nothing ->
          "There is no oldest cat!"
        Just house ->
          "The oldest cat is "
          ++ getCatName (getCatFromHouse house)
          ++ ", is "
          ++ show (getHumanAgeOfCatFromHouse house)
          ++ " equivalent human years old"
          ++ " and it lives in Number "
          ++ getHouseNumber house
```

Phew! We've covered a **lot** of code in this chapter.

The two helper functions `getCatName` and `getHouseNumber` use pattern matching to grab out the name of a cat and the number of a house (and make sure they're a string by using the `show` function).

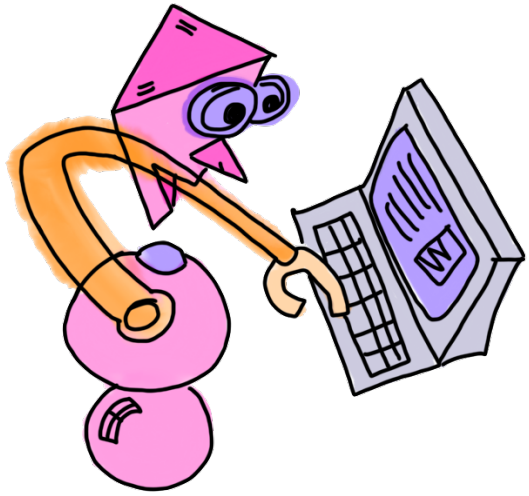
The `main` function uses a definition called `oldest` which uses a `case` expression on the application of `findOldestCatHouse` to `street`, which returns a value of type `Maybe House`. The `case` expression is used to match on either the `Nothing` value to report an error, or the `Just` value, in which case it takes the house out of the `Just` value constructor with pattern matching and creates a nice descriptive sentence about the oldest cat, its age in human equivalent years, and where it lives.

14.10 Homework

Your homework is to write a definition for a name and value whose type is `String`, and a program that prints that `String` using `putStrLn`. Make sure you can do it from memory without looking it up, and write out the types for all definitions in your program.

15 Basic Output

Covers: writing programs to create basic output



15.1 Setup Your Environment

Let's start writing some software! At this point, you should find out how to use an interactive Haskell environment because we'll be using it to check types and run code. An internet search will help you to find this. There are countless tutorials on how to set up and install this software, and it has a tendency to change, so we won't repeat them here, but if you're using GHC, then you will already have the

interactive environment GHCi installed, which is what we'll be using. A good start is to go to `http://www.haskell.org/` and to use the **stack** environment because it sorts out a lot of the common problems with setup and installation.

In particular, when you write your Haskell files in a text editor, you load them into GHCi with the `:load filename` (or `:l filename`) command and then you can run `main` and other functions by typing `main` and pressing the return key. You can also check types by using the `:type` (or `:t`) command in front of an expression, and when you've changed your code, you can use `:reload` (or `:r`) to reload the file, which gives a pretty nice workflow for constantly type-checking your work.

You'll also need a text editor. Atom, Sublime Text or Textmate are reasonably good ones, depending on if you're on Windows or Mac.

In a later update to this tutorial, we will most likely address setting up Haskell and which tools to use.

15.2 putStrLn, print and String

We begin by taking a number of simple, graded exercises. By now, you're familiar with recognising the `putStrLn` & `print` functions and `String` values and their types. Doing these exercises, and then revising them later will anchor these things in your long term memory.

15.3 Ways To Solve Problems

To do them, we'll run through some ways to think about solving them together, then at the end you will get to solve similar problems on your own. You should **not** look at the example run-throughs when you're doing your own programs, otherwise you won't build your own real-world usage understanding. If you have to look, that's fine, however once you've finished, you should do that exercise again from the start without looking.

These exercises may seem stupidly simple at first, but they're designed to get you to think in a way that will pay off as the programs get more complicated and larger with time.

15.4 Guided Exercise 1: Display Hello

Task: Write a program to print "Hello" on the screen.

We can immediately see we'll need the `String "Hello"`, so let's create a definition for that, and its type:

```
helloString :: String
helloString = "Hello"
```

If we didn't already know the type of `"Hello"`, we could ask GHCi by typing `:t "Hello"` at the prompt, and it will tell us `"Hello" :: [Char]` and as we know, `String` is a type synonym for `[Char]`. This can be very helpful for working out the types

of expressions and values so we know what will work with what.

So now we have our `String`, we want to be able to print it on the screen. This will be an `IO` action, and as we know, Haskell programs always start with the `IO` action `main`, so we will need to make a definition for that. The type of `main` has to be `IO ()`.

So, on the one hand we have `helloString` whose type is `String`, and on the other, we need a definition for `main` whose type is `IO ()`.

So we want a function whose type is `String -> IO ()`. We could use the **hoogle** Haskell search (at `https://www.haskell.org/hoogle/`) to find this for us by typing in the type signature to its search feature, but we happen to already know of a function whose type matches this, because we've seen it many times by now: `putStrLn` has type `String -> IO ()`, and we know that means if we apply a `String` value to it by putting it to the right of it, together they will be an expression whose type is `IO ()`. We also know how to make definitions: you write a name on the left, an equals sign, then an expression on the right.

So, we can join all of this information up, and finish our program:

```
helloString :: String
helloString = "Hello"

main :: IO ()
main = putStrLn helloString
```

15.5 Guided Exercise 2: Display the Sum of Two Numbers

Task: Write a program to add the number 3029 to 2938 then print the answer on the screen by itself.

Ok, we'll use a top-down approach this time. We know we'll need to have a program that prints a number on the screen.

We're very familiar with printing numbers on the screen by now. We can either use `putStrLn` with a `String` version of the number, or the `print` function which takes any instance of `Show` (which `Integer` is), so let's write our program "as if" we already had the addition expression that evaluated to a single number. To do this, we can "dummy in" a definition of a single number (let's use `0` for now):

```
theAnswerNumber :: Integer
theAnswerNumber = 0

main :: IO ()
main = print theAnswerNumber
```

If you compile this and load it into GHCi, it will work just fine.

Now, the only thing that remains is to work out how to change that expression from just `0` to an expression that will add the numbers together, and we know about the `(+)` operator which takes two numbers, and returns their sum. Let's use it:

```
theAnswerNumber :: Integer
theAnswerNumber = 3029 + 2938

main :: IO ()
main = print theAnswerNumber
```

15.6 Guided Exercise 2: Display the Product of Two Numbers

Task: Write a program to print the product (multiplied value) of `33` and `398`.

Again, we know the type of `main` is `IO ()`, and we know the type of `print` is `Show a => a -> IO ()`, so we'll feed `print` a single value to make an expression which we can define as `main`.

```
main :: IO ()
main = print 0
```

Now we have this, what if we just replace the value `0` with `33 * 398`? Well, in Haskell we know that a space between things will mean we want function application, and we know that this is very high precedence, and it'll apply from left to right, so `print 33 * 398` would mean the same thing as `(print 33) * 398`, and if we try that, Haskell will give us this type error, because it'd be trying to apply `(*)` to an `IO ()` value...: `No instance for (Num (IO ())) arising from a use of '*'`.

That means it first bound `33` to `print`'s argument, to give an expression of type `IO ()`, and then tried to pass **that** `IO ()` value into `(*)` as one of its arguments, which didn't work, because the type of `(*)` is `Num a => a -> a -> a`, not `IO ()`, which does not have an instance of the `Num` typeclass. That's what that error message says.

So, we need to use brackets to turn `33 * 398` into a single expression which can be passed as the one argument to `print`. Once we do that, it works, and that's our program done:

```
main :: IO ()
main = print (33 * 398)
```

Now it's your turn!

15.7 Reader Exercise 1

Task: Write a program that prints `"This sentence is false"` on the screen. Once you've done this, make it say `"No it's not"` instead. Once you've done this, change it to say `"One plus two is not seven"`.

15.8 Reader Exercise 2

Task: Write a program that prints the number `20938` on the screen.

15.9 Reader Exercise 3

Task: Write a program that adds the number of whatever the current month is right now (1 is for January, 2 is for February etc.) to the current year, and prints it on the screen. (Note: it needn't actually generate or get the current month, you just write it in as a number yourself).

15.10 Reader Exercise 4

Task: Write a program that multiplies your current age with your mother's current age and prints it on the screen. (Just write the values you know for these ages in as numbers, you don't need to make the computer actually get the numbers from the user).

15.11 Reader Exercise 5

Task: Write a program that adds the numbers 5, 7, 8 and 9 together and prints the result on the screen.

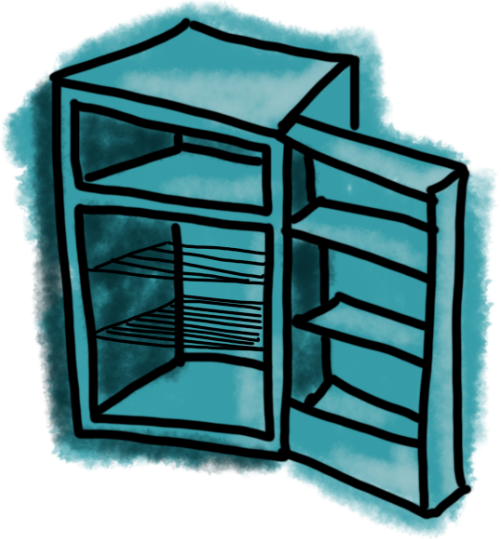
15.12 Reader Exercise 6

Task: Write a program that subtracts 999 from 1098 and prints it on the screen.

16 Fridge, The Game

This is one of the simplest horror games ever. The original idea was by Peter Halasz of `http://becauseofgames.com` who created it in the programming language **C** when he was learning how to program.

We're going to use this little game to learn how to get some input from the user, to sequence chunks of IO code together, and also to see a way to make a program continue until the user wants to stop.



Here is the listing. We explain it below.

```haskell
main :: IO ()
main = do
  putStrLn "You are in a fridge. What do you want to do?"
  putStrLn "1. Try to get out."
  putStrLn "2. Eat."
  putStrLn "3. Die."
  command <- getLine
  case command of
    "1" ->
        putStrLn "You try to get out. You fail. You die."
    "2" ->
        do
          putStrLn "You eat. You eat some more."
          putStrLn "Damn, this food is tasty!"
          putStrLn "You eat so much you die."
    "3" ->
        putStrLn "You die."
    _   ->
        putStrLn "Did not understand."
  putStrLn "Play again? write y if you do."
  playAgain <- getLine
  if playAgain == "y"
  then main
  else putStrLn "Thanks for playing."
```

This program uses a `do` block, which is something we haven't seen before. This allows you to join many actions together into one single action. There are two main things to know about `do` blocks, which we'll cover below.

16.1 Do Blocks with IO

Firstly, all of the actions in an `IO` do block are executed in the order they're written. That is, they're sequenced together. Secondly, using "`<-`", you can connect an **inner value** from an `IO` action on its right to a variable on its left. It's worth noting that you can only use this `<-` syntax within a `do` block, though.

You'll also notice that we have a `case` expression. We can use any Haskell expressions we like in a `do` block as long as they result in an action of the same type as the `do` block's type.

Anyway, let's see these things in action with two tiny example programs, one for `do` blocks combining and sequencing `IO` actions, and one for gathering input from the user.

```haskell
main :: IO ()
main =
  do
    putStrLn "Hello"
    putStrLn "There"
```

Ok, so this program will first print `Hello` on the screen, then it will print `There` on the next line. The do block takes one or more actions, and packs them into a single action. The type of that do block above is `IO ()`, just like `main`, and in `IO`, these actions will be sequenced one after the other as we've written them down the page.

And now, getting some input from the user:

```
main :: IO ()
main =
  do
    putStrLn "What is your name? "
    theName <- getLine
    putStrLn ("You said your name is " ++ theName)
```

Ok here we're sequencing **three** actions together. We can get input from the user in Haskell with `getLine`. When this program is run, once the first line has been output, it will wait for the user to put some text in and press return. Once they do that, it will have bound that text into the `getLine` action, pulled that value out as `theName`, and printed out the last line which includes that text the user entered!

If we look back at our original program, and look at the line `command <- getLine`, you'll see `<-` is there to pull the `String` from the `getLine` action and set the variable "`command`" to its value. The type of `getLine` is `IO String`, which means it's an `IO` action that "contains" a `String` when it's executed. When we use `<-`, we can think about it like it's extracting the `String` value from the action (note that this applies only when we're in an `IO` action, though).

16.2 Do Block Nesting

Notice, also, that you can put `do` blocks within `do` blocks. This is called **nesting**. We're doing this by having another `do` block in the "`2`" branch of the `case` expression.

16.3 Whole-Program Recursion

This game has two sections. First it tells the user their options and asks for their input with `getLine`, and then depending on what they wrote, it tells them what happened. Next, it asks if they want to play again, and if they do, it runs the whole thing again by calling `main`. This is recursion of the whole program. The program is an `IO` action, and `do` blocks allow us to compose `IO` actions, so it's perfectly fine to have the whole program at the end recursively.

One last thing to note about `do` blocks, though, is that they must always end with the same type as the whole `do` block. So, becuase ours ends with an `if` expression whose resultant type is `IO ()`, which is the type of the `main` function itself, it will work just fine. We'll see more examples of `do` blocks in later chapters.

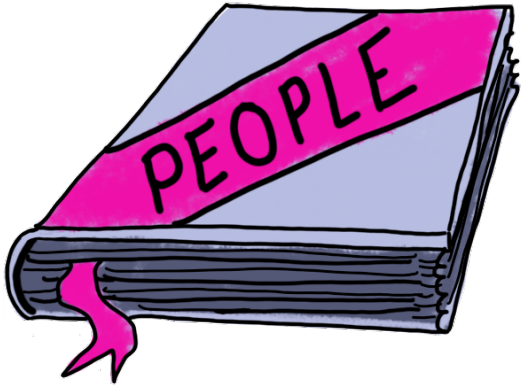
16.4 Homework

Homework is to go for a walk with a pad and pen and write a program to add up a few of the numbers on number plates of cars in your street (assuming there are lots of cars around, if not, pick something else where there are lots of numbers). Add them up using the `(+)` function, and use either `print` or `putStrLn` with `show`. Do this as many times as you need to so that you know how to do it without looking it up. You will probably need to do this in a few different sessions to fully anchor it in your memory. Also, write a function that prints out a greeting.

You'll notice that we're giving you lots of repeated homework with `putStrLn`, `print`, `show` and the other basics. That's because we want to make sure you can do it very well. Varied repetition, or practice, is the key to getting very good with a skill.

17 The People Book

In this chapter, we'll see some code for working with our own super-simple address book, and in the process introduce an extremely useful variety of functions called **higher order functions**. We'll also dig a bit deeper into **sum** and **product** types (or **algebraic data types** as they're generally known as together), and introduce **records**, another way to work with data within such data types.



Imagine you were building up a list of your favourite people from the subject areas of maths and computing. (I know, what a silly suggestion!) So, what kind of information would you want to write down about them? Let's just pick a couple of things about them to keep track of.

17.1 Models of Data

A data model is a set of data types or data that describes something. We've actually been doing data-modelling the whole book. When we say that we're going to use an `Integer` value to represent someone's age, for example, we're doing **data-modelling**. In our case right now, we're about to be modelling people. So, what data would make up a person?

Well, because a person can have many pieces of information about them (or we could call them fields or attributes), we need a way to build a single type out of a combination of other types. To do this in Haskell, we use a **product type**, as we've briefly seen before. Here's an example of one of these **product** types:

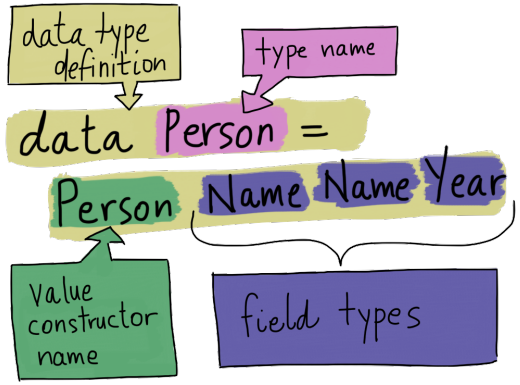
```
type Name = String
type Year = Int
data Person = Person Name Name Year
  deriving (Show)
```

You're probably wondering about some things here. What is "deriving"? Why is `data` being used without the `|` symbol, and why is `Name` written twice? We'll go through this now.

Firstly, we can see that we have a type alias for `Name` as `String`. We also have one for `Year` as `Int`. Great, there's nothing new there. We know that just says these different type names can be used for those types, and Haskell will know what we mean. As you know, these are called type aliases or type synonyms.

17.2 More on Data Types

What about `data Person` though? Well, this is how we define `Person` to be an **algebraic data type**, as we mentioned above. The `data` keyword tells Haskell we're creating our own fresh new type. We've seen these before, but let's go through it in more detail to understand it better.



The part to the left of the `=` symbol is the **type name**. Once our type is created, this name can be used in places where types can go, for example in type annotations. This name is `Person` in the example above. If we'd written `data Muppet = HappyMuppet Name` instead, then `Muppet` would be the type name instead.

Next we'll look to the right of the `=` symbol. We see `Person` again. This is a **value constructor**. When we create a type like this, Haskell creates us a function (or just a value if there are no type fields) for

each of the values the type describes. In our case, we just have the one, which is `Person`. Looking to the right of this, we can see the types that make up this `Person` value, and the value constructor.

If we'd created the more complicated data type of `data SizedPerson = TallPerson Name | ShortPerson Name`, then we would have **two** value constructors: `TallPerson :: String -> SizedPerson` and `TallPerson :: String -> SizedPerson`, just to show you how it looks when there are sum **and** product types in the one algebraic data type.

Anyway, it's called a product type because a single piece of this type of data is a **product** of more than one piece of data of these types. These pieces of data are often called **fields** of the data type. Again, if we'd written `data Muppet = HappyMuppet Name`, then `HappyMuppet` would be the **value constructor** (also sometimes called the **data constructor**).

So, after we've defined this type, Haskell will have defined a new value constructor function for us automatically `Person :: Name -> Name -> Year -> Person`. Notice that the return type of this function is `Person` as well as the function being named `Person`. If we go back to our muppet example as a contrasting example, we can see that the type of the data constructor would be this instead: `HappyMuppet :: Name -> Muppet`.

17.3 Making Our Types Show

There is also this `deriving (Show)` line after `Person`. That is for us to tell Haskell that we'd like it to create an easy to print version of this data type for us, automatically creating an instance of the `Show` typeclass for this data type. When we use `show` on it, it will just print it out like it's written in the code.

17.4 Building Our First Value

Back to `Person`, though, those `Name` fields... why **two** names? Is it first and last names? If so, which is which? Let's see an entry for a person. Maybe that will clear it up:

```
-- the famous mathematician
-- Blaise Pascal
blaise :: Person
blaise = Person "Blaise" "Pascal" 1623
```

Ah, so the given name comes first, and the family name goes second. But, what is `Year` supposed to represent here? Is it the year of birth? of death? of something else?

17.5 Records

Don't you wish there was some way we could see exactly what the types were supposed to mean **inside** of the **product type** itself? Well,

it turns out there **is**. It's what's called **record syntax**, which is simply a way to let us name the fields as we specify a data type. Here's what the `Person` type looks like using **record syntax**:

```
type Name = String
type Year = Int
data Person = Person
    { personFirstName :: Name
    , personLastName :: Name
    , yearOfBirth :: Year }
  deriving (Show)
```

Okay, there are names for the fields now, so it's clearer what each means. Also, Haskell automatically makes what is called a **getter** function for the fields, and it also allows us to use something we'll see called **record update syntax** for making new data based on existing data. So, `personFirstName` is a function of type `Person -> Name`, which means we pass it a `Person` and it gives us back the first name, and so on for the other fields. How about constructing a `Person` now? What's that like? Well, it can work like this:

```
blaise :: Person
blaise =
  Person { personFirstName = "Blaise"
         , personLastName = "Pascal"
         , yearOfBirth = 1623 }
```

However, we can still build a `Person` the usual way, and all the old things work as before.

Something new, though, is that we can also easily build new records out of others by doing the following:

```
traise :: Person
traise = blaise { personFirstName = "Traise" }
```

This is called **record update syntax**. This one creates a person whose data looks like this: `Person {personFirstName = "Traise", personLastName = "Pascal", yearOfBirth = 1623}`. Note that we can now "set" and "get" the data for fields in any order we like. We can set more than one field at once, too.

Let's look at some more people:

```
people :: [Person]
people =
  [ Person "Isaac" "Newton" 1643
  , Person "Leonard" "Euler" 1707
  , Person "Blaise" "Pascal" 1623
  , Person "Ada" "Lovelace" 1815
  , Person "Alan" "Turing" 1912
  , Person "Haskell" "Curry" 1900
  , Person "John" "von Neumann" 1903
  , Person "Lipot" "Fejer" 1880
  , Person "Grace" "Hopper" 1906
  , Person "Anita" "Borg" 1949
  , Person "Karen" "Sparck Jones" 1935
  , Person "Henriette" "Avram" 1919 ]
```

17.6 Finding a Person from the List

Let's see how we'd find a particular person, say, the first person whose birthday is after 1900 in the list. First, we need to make sure we've imported the `Data.List` module, because we'll be using the `find` function from that module. At the top of our code file, we make sure the import is present:

```
import qualified Data.List as L
```

Then, we can find our person with this definition of an expression:

```
firstAfter1900 :: Maybe Person
firstAfter1900 =
    L.find (\(Person _ _ year) -> year >= 1900) people
```

The `find` function has the following type signature `L.find :: Foldable t => (a -> Bool) -> t a -> Maybe a`. This means it takes two arguments: a function from some type of value called `a` to a `Bool` known as the predicate, and a `Foldable` of that same `a` type. Our `Foldable` instance will be on list, because we have `people :: [Person]` as our `Foldable t => t a` value.

Essentially what the function does is apply the predicate to each of the items until one returns `True` in which case it returns that particular item wrapped in `Just`, otherwise it returns `Nothing`. It's not the most efficient way to find things because of the way lists are constructed, but it will be fine for our purposes here.

Also to note here is the way we're using the `Person` constructor to pattern-match out the parts of the `Person` as it gets fed into the `find` function, with our predicate: `(Person _ _ year) -> year >= 1900)`. The two underscores simply throw those particular fields away because we're not interested in them, and we just match out the year as the variable `year` which we then compare to `1900`.

Instead of doing it like that, we could also have written it like this, which is a bit more flexible because it doesn't depend on the field ordering in the data type:

```
firstAfter1900' :: Maybe Person
firstAfter1900' =
    L.find (\person -> yearOfBirth person >= 1900) people
```

17.7 Filtering out People in a List

Let's see how we'd find the sub-list of people whose name begins with L, using recursion and the list above:

```
firstNameBeginsWithL :: Person -> Bool
firstNameBeginsWithL p =
  case personFirstName p of
    'L':_ -> True
    _     -> False

makeNewListWithOnlyLPeople :: [Person] -> [Person]
makeNewListWithOnlyLPeople [] = []
```

```
makeNewListWithOnlyLPeople (x:xs)
  | firstNameBeginsWithL x =
      x : makeNewListWithOnlyLPeople xs
  | otherwise =
      makeNewListWithOnlyLPeople xs

peopleThatBeginWithL =
  makeNewListWithOnlyLPeople people
```

The `firstNameBeginsWithL` function takes a `Person` as the variable `p`, gets the first name with the `personFirstName` getter function, then we have a `case` expression on that.

If the first name is a `String` beginning with the letter `L`, it will match `'L':_` because `(:)` is, as we know, a value constructor that matches any list and pattern-match splits it into its head and tail. This returns `True`, otherwise the "`_`" pattern will pattern-match on anything else, which means we return `False`.

Next we'll look at the `makeNewListWithOnlyLPeople` function, which is a reasonably simple recursive function using **guard patterns**. Rememeber that **guard patterns** work by Haskell matching the first expression that evaluates to `True`, then returning the expression on the right of the corresponding `=` symbol. We pull the `Person` list into head and tail as `x` and `xs` respectively using pattern-matchin again. If the `Person` in `x` has a first name that begins with `L`, we add it to the return list by using `(:)` to prepend it to the tail of the list (`xs`) with the function applied to it. If it doesn't begin with `L`, we simply apply the function to the tail of the list.

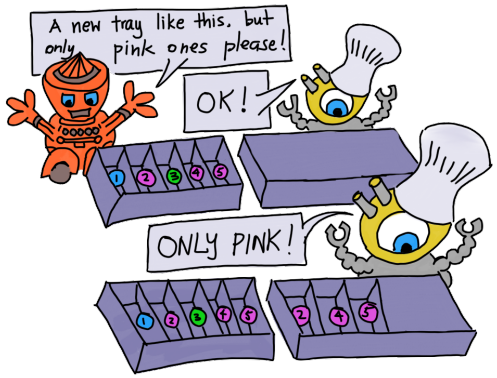
You might notice that `makeNewListWithOnlyLPeople` is **using** the `firstNameBeginsWithL` function as a kind of testing function. This type of function is called a **predicate** function in programming. It checks if something is true or not. What if we wanted to be able to swap out that function, and make a whole lot of different lists with people whose names started with letters other than `L`? Well, next we'll look at a general way to do just this.

17.8 A Note About List Efficiency

You might be thinking that this way of finding something is very inefficient. You'd be correct if you were thinking this! For small amounts of data, the list type is very handy and useful, and more than efficient enough. However, for much larger amounts of data, we would use have to use different functions and types if we wanted things to be fast and efficient. We'll see these in later volumes. As with everything, the context gives meaning to the content, so as the content changes (you get a bigger set of data), we must choose different ways of working with it (choose different context of functions).

Lists are very good for certain things, such as for representing data that is added to at the front. They are also quite easy to write code for, so they're good for beginners to look at first, such as yourself. As you learn programming more you'll start to get an appreciation and understanding of the different types for storing data, and when it's good to use each.

17.9 Higher Order Functions: filter



What we just saw is a very common pattern in Haskell: we take a testing function that returns a `Bool` value (otherwise known as a **predicate**) and a list of the same type of items that the **predicate** takes, and we return a new list of items that resulted in `True` when "tested" against the predicate. It's so common that there's already a built-in higher-order function for it in Haskell called `filter`:

```
filter :: (a -> Bool) -> [a] -> [a]
```

A **higher-order function** is a function that takes one or more functions as argument(s). This term also applies to functions that return functions, but because of the way functions work in Haskell, that's so common and easy that we usually don't count it.

Let's see how rewriting our "only L people" function using `filter` simplifies it:

```
makeNewListWithOnlyLPeople' :: [Person] -> [Person]
makeNewListWithOnlyLPeople' xs =
  filter firstNameBeginsWithL xs
```

Here, `xs` is matched to the list of type `Person`, and we pass it to `filter`, along with our predicate (`firstNameBeginsWithL`). This version does exactly what our previous function does, but with a lot less code to read and write. Using these built in common function like `filter` and the others we'll see later is really useful because it saves us having to reinvent the wheel each time we want such common functionality. It also gives us a common language to talk about these things with other Haskell programmers.

17.10 Some Eta Reduction

We can further simplify the definition by removing the `xs` from both sides of the equals sign!

```
makeNewListWithOnlyLPeople'' :: [Person] -> [Person]
makeNewListWithOnlyLPeople'' =
  filter firstNameBeginsWithL
```

That is, `filter` usually takes two arguments: a function of type `(a -> Bool)` (any type at all to `Bool`), and a value of type `[a]` (a list of that same type), then returns a value of type `[a]` (another list of that same type). If we were to supply it with both arguments, it

would return us value of type `[a]`, but if we only supply the first one (the predicate from `a -> Bool`), then we'll end up with a function from `[a]` to `[a]`!

The technical name for this process of getting rid of these variables that are repeated on the right hand side of the inside and outside is called **eta reduction**.

Here's another example of it:

```
plus num1 num2 = num1 + num2

plus' num1 = (num1 +)

plus'' = (+)
```

All three of these functions work in the same way. The `(+)` function already takes two arguments, which as we know in Haskell means it is actually two nested functions. Let's look at yet another way to do the same thing, this time with lambdas:

```
add = \x -> (\y -> x + y)

add' = \y -> (+y)

add'' = (+)
```

In each step, we're simply removing one of the unnecessary variables from our function definition, because `(+)` is **already** a function itself, so by the end, all we're doing is effectively saying that `add''` is simply the `(+)` function.

17.11 Using filter

Getting back to our original functionality, let's look at a different way to write the whole function:

```
-- don't get confused, c is not the letter c here
-- it's a variable name, holding the Char value
-- we're matching on
firstLetterIs :: Char -> String -> Bool
firstLetterIs c ""    = False
firstLetterIs c (x:_) = c == x

firstNameBeginsWith :: Char -> Person -> Bool
firstNameBeginsWith c p =
    firstLetterIs c firstName
  where firstName = personFirstName p

peopleThatBeginWithL :: [Person]
peopleThatBeginWithL =
  filter (firstNameBeginsWith 'L') people
```

We have `firstLetterIs`, a more general function that takes a `Char` and a `String` and returns `True` if the first letter of the `String` is the passed in `Char` value. The beauty of this function is if we decide we want to use a different letter, we just have to change the one spot in the code.

Then there's the `firstNameBeginsWith` function that gets the first name of the passed in `Person` and matches its first letter against a passed in `Char` value by using the `firstLetterIs` function.

Finally, we use `filter` along with the partially applied function `firstNameBeginsWith 'L'` and the people list to create a `Person` list value defined as `peopleThatBeginWithL`.

It might be pretty clear to you now how we can easily build a list by filtering on the first name beginning with any character we like, and it should be reasonably easy to see how you could create a list of people whose last names start with a different letter. (For example, `filter (firstNameBeginsWith 'H') people`).

17.12 Higher Order Functions: map

Now we're going to take a look at something we'll need later on. We may want to get the last name of a `Person`. We know how to do this for one `Person` just fine. Let's say the person is `blaise`, then we'd write `personLastName blaise`. That's pretty straightforward.

Well, what if we actually wanted to get a whole **list** of first names from a whole list of people? What would we use? Well, given what we know about recursion, we'd probably write it something like this:

```
peopleToLastNames :: [Person] -> [String]
peopleToLastNames []     = []
peopleToLastNames (x:xs) =
  personLastName x : peopleToLastNames xs
```

Study this little function well! What we're seeing is a function with two definitions, as usual. It's looking like some standard recursion. The first definition simply says if the `[Person]` passed in is the empty list of `Person`, return the empty list of `String`.

The second definition is where most of the work takes place. This first pattern matches the head of the list into `x` and the tail into `xs`. So, `x` will be a `Person`, and `xs` will be a `[Person]`. It then returns applying `personLastName` to `x` which gives us a `String`, then prepends this using `(:)` to the result of calling the whole function again on the tail of the list (recursively).

Next we can imagine what it'd be like if we wanted a kind of general function so we weren't locked in to **only** using the `personLastName` function. Let's see what an equivalent first name function would be like, first:

```
peopleToFirstNames :: [Person] -> [String]
peopleToFirstNames []     = []
peopleToFirstNames (x:xs) =
  personFirstName x : peopleToFirstNames xs
```

Not much changes, does it? We've really only changed the function that gets called on each `Person` value to turn it into a `String` value. What if we made a general function and let the programmer using the function pass in their own function of type `Person -> String`, that way we could make this quite general:

```
mapPeople :: (Person -> String) -> [Person] -> [String]
mapPeople f []     = []
mapPeople f (x:xs) =
  f x : mapPeople f xs
```

Ok, notice we've added another function argument at the front — it's `f`, which is the function of type `Person -> String` we're passing in — and all our function definitions now have an added `f` parameter and variable. Also notice we're using it by appling it to `x` before using `(:)` with our recursive function application of `mapPeople` at the end of the last line which also has to have the `f` argument, as it's now a required argument to our function.

This is now quite general. We can use this with first names or last names. Let's see the redefined version of these functions now:

```
peopleToLastNames :: [Person] -> [String]
peopleToLastNames people =
  mapPeople personLastName people

peopleToFirstNames :: [Person] -> [String]
peopleToFirstNames people =
  mapPeople personFirstName people
```

That's starting to look **very** nice and compact. We now know, though, that we can **eta reduce** these functions by getting rid of the people argument, as follows:

```
peopleToLastNames :: [Person] -> [String]
peopleToLastNames = mapPeople personLastName

peopleToFirstNames :: [Person] -> [String]
peopleToFirstNames = mapPeople personFirstName
```

Nice. However, it's time to let you in on a secret. The `mapPeople` function already exists in Haskell, as an even **more** general function called `map`. This one works on lists of **any** type of value at all.

Let's see its type signature:

```
map :: (a -> b) -> [a] -> [b]
```

This takes two arguments: a function from anything `a` to anything `b`, a list of those `a` values, and returns a list of those `b` values. For us, this means we'd want these `a` and `b` values to be `Person` and `String` (we call this **specialisation**). Pay careful attention and note that where Haskell has written `a` and `b` in type signatures, that doesn't mean that those types **have** to be different, only that they **can** be different, if you'd like.

To illusrated this, if you want to `map` from `String` to `String`, or the same type to the same type, there's nothing stopping you. For example, here's a function that maps from a list of strings to their reverse string counterparts, using the `reverse` function: `reverseMap = map reverse :: [String] -> [String]`.

So, anyway, we could have just written our `mapPeople` function like this:

```
mapPeople :: (Person -> String) -> [Person] -> [String]
mapPeople = map
```

Or, we could just have used `map` instead of `mapPeople`.

So, given we have a list of people called `people`, we'd create a list of their last names like this:

```
lastNames :: [String]
lastNames = map personLastName people
```

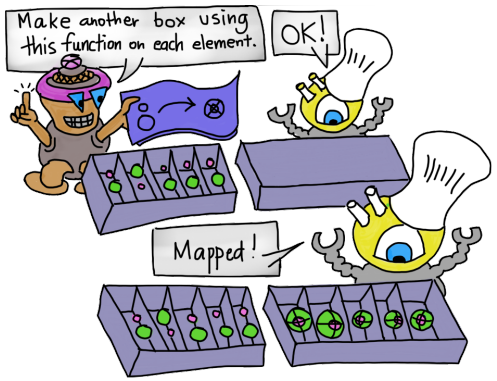
Very nice and compact.

So, the higher order function `map` takes a function and a list. It gives a new list with the function applied to each element of the list. Let's see how `map` could be implemented:

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

As you can see, it's very similar to the function we had for getting the last names of people.

Here's a picture that might help you understand what map does visually:



You get a whole new list with new items that have been created by applying the function to the original elements. The original list and items are still there, unchanged.

This is important: Haskell values are almost never modified, they're always created afresh, or referred to if they're identical.

You might be tempted to think of it a bit like the function is "doing something" to each element, especially with these higher order functions, but it's not, it's looking at the original element and using the passed-in mapping function to make an entirely new element for the new list, leaving the original untouched. This is very good, because if two functions are referring to one value, the last thing you want is for that value to be changed by one of the functions without the other one realising it. Luckily, this can't happen in Haskell. This is because Haskell has **purity**, which means functions cannot work on things other than their arguments.

17.13 Higher Order Functions: sortBy

Ok, so now we have our last names in our `lastNames` variable, perhaps we might want to sort them. First, we need to make sure we've imported the `Data.List` module, because the sorting functions are in that module. At the top of our code file, we make sure the following is present:

```
import qualified Data.List as L
```

Before we get to these functions for sorting, we need to know a little about the `Ord` **typeclass**. This is for types that can be ordered. `Ord` provides the functions `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `min` and `max`. These functions allow comparison between two values in various ways. You should investigate their types using **hoogle**. There is also a **sum type** called `Ordering` that this typeclass uses that has the values `LT`, `GT`, and `EQ`, which represent less-than, greater-than and equal-to respectively. The `compare` function returns this type, and sorting functions use the `compare` function and the `Ordering` type to do their work.

Right, so now we can have a definition for our sorted last names using the `sort` function from the `Data.List` module, whose type signature is `Ord a => [a] -> [a]`. It takes a list whose element type must be instances of `Ord` (for comparing and ordering, remember), and returns the sorted version of that list.

```
sortedLastNames :: [String]
sortedLastNames = L.sort lastNames
```

This will be sorted alphabetically. What if we wanted it in reverse? Well, there is a `reverse` function in Haskell whose type is `[a] -> [a]` that will reverse any list at all. Instead of using this, though, we're going to see how to use a function called `sortBy` that takes a function like `compare`, instead.

Firstly, `sortBy` has a type of `a -> a -> Ordering -> [a] -> [a]` and `compare` has type `a -> a -> Ordering`, so you'll notice that the `sort` function is effectively the same thing as `sortBy compare`. To reverse the order, we can simply provide a function to `sortBy` that returns the opposite `Ordering` that compare would return, and we can do that by swapping the arguments to `compare`:

```
reverseSortedLastNames :: [String]
reverseSortedLastNames =
  L.sortBy (\x y -> compare y x) lastNames
```

This definition is more efficient than doing `reverse (sortBy compare lastNames)`, because it only has to go through the list once. For our small data set, this is not going to be a problem. It would matter with a very large list, though. In that case, though, a list would probably not actually be the best data structure to use.

We can see there that we've used a lambda of two arguments to **flip** the order of `compare`'s arguments. This has the intended result of a reverse-sorted list of the lastNames.

There's an arguably better way to do this than use a lambda, though. Haskell has a commonly-used function called `flip` that

works with any function of two or more arguments. Pass it any 2-argument function, and it'll return you a function that works exactly the same, but has its arguments swapped. So here's an alternate way to write `reverseSortedLastNames`:

```
reverseSortedLastNames' :: [String]
reverseSortedLastNames' =
    L.sortBy reverseCompare lastNames
  where reverseCompare = flip compare
```

At this point, how we get the list of `firstNames` should appear as no surprise.

```
firstNames :: [String]
firstNames =
  map personFirstName people
```

Ok, but what if we wanted to sort the people list itself by some field of each person? Well, `Data.List` has a `sortOn` function whose type is `Ord b => (a -> b) -> [a] -> [a]`. It orders the new list by the result of some function `a -> b` applied to each element.

Let's say for our purposes we want to create a list of people sorted by their first names. The function `personFirstName` fits the `a -> b` type perfectly for our purposes, as its type is `Person -> String` and we want to sort on the first name `String`.

```
sortedPeopleByFirstName :: [Person]
sortedPeopleByFirstName =
  L.sortOn personFirstName people
```

Now let's see a function that takes a `year`, and a `person`, and works out how many years ago from that year that person was born.

```
yearsSinceBirthAtYear :: Year -> Person -> Int
yearsSinceBirthAtYear y p = y - yearOfBirth p
```

We map `y` to the comparison year, and `p` to the passed in person, then apply the `yearOfBirth` function to the person and subtract that from the comparison year. If we wanted to get this across all the people, we could map it as a part-applied function, say for `2012`:

```
allYearsSinceBirthAt2012 :: [Int]
allYearsSinceBirthAt2012 =
  map (yearsSinceBirthAtYear 2012) people
```

The type of `yearsSinceBirthAtYear :: Year -> Person -> Int` means if we apply one argument to it (`2012` in this case), we'll end up with a function `(Person -> Int)` that is locked to compare with `2012`, takes a single argument (a `Person`) and replies with the number of years difference between `2012` and that person's birth year.

And now a function that shows the earliest year of birth for the people on our list. This uses the `minimum` function which will work on lists containing instances of `Ord`. Actually it's a very general function, because it will work not only on list, but any instance of `Foldable`. There's also a `maximum` function that gets the highest ordered value, too. Let's see their type signatures before we proceed:

```
minimum :: (Ord a, Foldable t) => t a -> a
maximum :: (Ord a, Foldable t) => t a -> a
```

These functions have **two** typeclass constraints on them. This says that `t` must be an instance of the `Foldable` typeclass, but also that `a` must be an instance of the `Ord` typeclass. Note that you cannot pass an empty list into these functions. You must only pass a list that has at least one item in them.

Luckily for us, `Int` has an instance for `Ord`, and list has an instance for `Foldable`, so `minimum` will work perfectly for us:

```
earliestYearOfBirth :: [Person] -> Year
earliestYearOfBirth people =
  minimum (L.map yearOfBirth people)
```

17.14 Removing Parentheses With The ($) Function

In Haskell, we prefer not to use so many parentheses. There is a higher order function called `($)` that will take a function on the left of it, and some expression on the right, and apply the function to the expression. It has an extremely low precedence, which means it will pretty much be applied last of all. This is basically the same effect as having parentheses wrapped around the expression on the right. Let's see how the function above can be written with the `($)` function.

```
earliestYearOfBirth' :: [Person] -> Year
earliestYearOfBirth' people =
  minimum $ L.map yearOfBirth people
```

Note that we can't get rid of the `people` variable, though, because the `minimum` function is wrapping the whole expression `L.map yearOfBirth people`.

Note also that the `($)` function is only for the cases where the parentheses would go right to the very end of the expression on the right.

17.15 Using minimumBy

Last of all, we want to find out which `Person` was born first out of our list of people.

```
bornFirst :: [Person] -> Person
bornFirst people =
    L.minimumBy compareBirthYears people
  where compareBirthYears x y =
          compare (yearOfBirth x) (yearOfBirth y)
```

Lots to explain here!

`Data.List`'s `minimumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a` function is another higher-order function: that is, it takes a function which is a comparing function `(a`

`-> a -> Ordering)`. It also takes a `Foldable` instance wrapping some "`a`" values of any type, and gives us the minimum one by using the comparing function on successive pairs of items.

Notice that the "`a`" type doesn't have to be an instance of `Ord` here. So long as the comparing function returns an `Ordering` and its two arguments are the same type, `minimumBy` will compile with no problem.

Here, we're using a `where` clause to locally **scope** the comparing function `compareBirthYears`, which simply takes two people, matches them into `x` and `y` respectively, and returns the application of the `compare` function on their `yearOfBirth` fields.

Because `compare` has the type `Ord a -> a -> a -> Ordering`, and the `yearOfBirth` fields are `Integer` and they are instance of `Ord`, compare can do the comparison, and this means `compareBirthYears x y` returns an `Ordering`, which means it's the correct type that `minimumBy` requires.

These higher-order functions such as `map`, `filter`, `fold` and now `minimumBy` and `maximumBy` might seem complicated at first, but with lots of practice in thinking through all the types of the functions concerned, they will become second nature to read, and then later, to write.

17.16 Homework

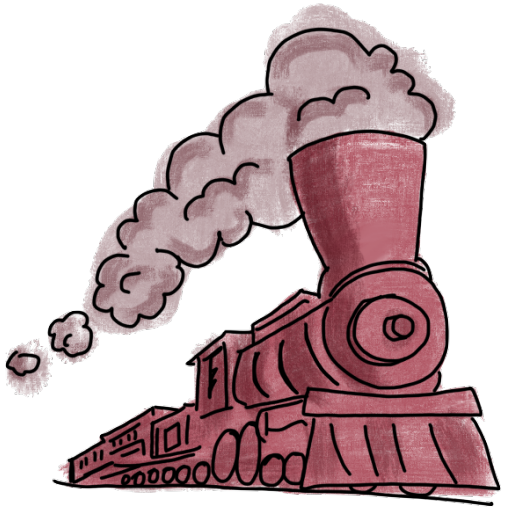
Your homework is to adjust the program by adding middle name (`middleName`) as a field to a `Person`, and adjusting all the functions and usage of functions as you go. Make up some middle names for these people. See if you notice that by using records, we've made it much easier to change our program. See if you can imagine how difficult it would be changing our program if all the functions were tied to the **shape** of our data type!

18 Times-Table Train of Terror

We're going to introduce you to a tiny toy educational game that introduces quite a few new functions and features of Haskell.

We won't be using any special **algebraic data types** in this program. In particular, our main type will be:

```
-- a game Level is simply a pair
-- of Integer values
type Level = (Integer, Integer)
```



18.1 Tuples or Pairs

A `Level` is simply a **pair** that has two `Integer` values. As this is a simple math game, each level will be a pair of numbers that represent the two numbers for a multiplication question. (For example, a level value of `(7,9)` would represent a question something like "What is 7 times 9?").

The next definition is where we build the levels for this game:

```
levels :: [Level]
levels =
    concat $ map pairsForNum [3,5..12]
  where
    pairsForNum num = zip [2..12] $ repeat num
```

18.2 Ranges and the zip function

Wow, there is a **lot** packed into this value expression. Let's pull it apart piece by piece.

The `levels` value is a list of `Level` values. It's defined using the `($)` function, which takes a function on the left, and applies it to the value or expression on the right.

In our case here, the value expression on the right is `map pairsForNum [3,5..12]`. The main new thing in this expression for us is `[3,5..12]`, which is what's called a **range**. The range

of numbers between 1 and 100, for example, would be represented as `[1..100]`. The range above, though, is every number between 3 and 12, in odd numbers. (So, 3,5,7,9,11).

So what we're doing, then, is mapping the `pairsForNum` function across this range. This function takes a number, calling it `num`, and "zips" the range `[2..12]` together with `repeat num`. Zipping is creating pairs from one item each of a number of lists as we'll see. The `repeat` function gives an infinite list of whatever its argument is. The `zip` function takes two lists, and builds pairs out of those lists, taking one item from each as it does. So, `zip [1,2,3] [4,5,6]` will create `[(1,4),(2,5),(3,6)]`. However, the `zip` function will stop when the first list runs out of values, so zipping an infinite list of repeated items with another that has only a few is just fine, as we're doing.

So what we'll end up with by using `map pairsForNum [3,5,..12]` is a list of lists of pairs of `Integer` values. This basically means we're combining each of the elements with each of the others. Then, we use `concat` to concatenate all the lists into one list. For example, `concat [[1],[2],[3]]` results in `[1,2,3]`, for example. In math, this is called a **cartesian product**. You don't have to know this, but later we'll see other, much simpler-looking, easier ways to do this. Unfortunately using them requires knowing more than we currently know, so that will have to wait.

The end result of this expression is that we have a list of 55 levels for our train of terror!

18.3 Determining the Level Number

Here's a function we'll use in our program to work out what number level the player is at:

```
levelNumber :: [a] -> Int
levelNumber remainingLevels =
    totalLevels - levelsLeft
  where totalLevels = length levels + 1
        levelsLeft  = length remainingLevels
```

We take a list of remaining levels, then subtract the number of levels left (using the length function) from the number of levels we started with plus one. As the player "moves up" the train, we'll be reducing the size of the list, so the remaining levels will get less, and the level number will go up. Notice from the type signature that we don't care what the element type of the list passed into it is, as long as it's a list.

The `main` function simply gives the player a greeting message, then starts the `trainLoop` function that handles all the game-play (by passing in the levels value).

```
main :: IO ()
main = do
  putStrLn "Suddenly, you wake up. Oh no, you're on..."
  putStrLn "The Times-Table Train of Terror!"
  putStrLn "Try to get to the end. We DARE you!"
  trainLoop levels
```

You can see we have our old friend the do block in action again, joining a whole bunch of IO actions together to form one.

In a moment, we come to the main game-play function, `trainLoop`.

18.4 The game loop

The function `trainLoop` has two definitions.

The first is for an empty list, which for our program means the player has won, because they got to the end of the train without failing. In this case, we declare their victory to them.

The second definition comes into play when there are still levels passed in. This means the game is still in play. Each time the player gets done with a level, we remove it from the list, then pass the rest of the levels back into the trainLoop function and start it again.

```
trainLoop :: [Level] -> IO ()
trainLoop [] =
  putStrLn "You won! Well done."
trainLoop remainingLevels @ (currentLevel : levelsAfterThisOne) =
  do
    let currentLevelNumber =
            levelNumber remainingLevels
        (num1, num2) =
            currentLevel
    putStrLn $
      "You are in a Train Carriage "
      ++ show currentLevelNumber
      ++ " of " ++ (show $ length levels)
    putStrLn "Do you want to:"
    putStrLn "1. Go to the next Carriage"
    putStrLn "2. Jump out of the train"
    putStrLn "3. Eat some food"
    putStrLn "q. Quit"
    activity <- getLine
    case activity of
      "1" ->
        do
          putStrLn $ "You try to go to the next carriage."
                   ++ " The door is locked."
          putStrLn "Answer this question to unlock the door:"
          putStrLn $ "What is " ++ show num1
                    ++ " times " ++ show num2 ++ "?"
          answer <- getLine
          if answer == (show $ num1 * num2)
          then do
            putStrLn "The lock is opened!"
            trainLoop levelsAfterThisOne
          else do
            putStrLn $ "Wrong. You try to open the lock,"
                     ++ " but it won't open."
            trainLoop remainingLevels
      "2" -> jumpingFutility
      "3" -> eatingFutility
      "q" -> putStrLn $ "You decide to quit."
                      ++ " Thanks for playing. Bah-Bye!"
      _   -> do
        putStrLn "That makes NO sense! Try again."
        trainLoop remainingLevels
```

The construction with an `@` symbol means the whole argument is matched as `remainingLevels`, and then, also, `currentLevel` is set to the first item of the list, and `levelsAfterThisOne` is set

to the tail of the list. When we use the `@` symbol like this, it's called an **as pattern**.

The `trainLoop` passes back an `IO` action, so we begin a very large `do` block. This `do` block has lots of different kinds of things in it, to get you familiar with more complicated looking `do` blocks.

Ideally, we would pull each of the sections into their own function. For now we'll look at it like this because it serves our purposes quite nicely to show you a larger `do` block.

Remember, each of these expressions will evaluate to an `IO` action (because the final value of this function's type is `IO ()`), and the `do` block simply connects them all up properly so they become one single `IO` action.

First, we set up some variables we'll need later using `let`. A `let` expression is another way we can define some locally scoped definitions. In a `do` block, name defined in a `let` expression will be available for the remainder of the `do` block. One of these is `currentLevelNumber` which uses the `levelNumber` function we just looked at, and the next line is pattern matching the `currentLevel` variable into two number variables `num1` and `num2`. Remember, `currentLevel`'s type is `Level`, which is `(Integer, Integer)`, so `num1` and `num2` will both be `Integer` values.

Then we print out a message explaining that the player is on a certain train carriage, using show to turn the number values into strings so they can be fed into `(++)` with the other strings.

Next, a menu is described, and we receive a line of text from the player with `getLine`, which goes into the variable called `activity`.

After this, we have a `case` expression that matches on the player's response. If they wrote 2, 3 or q, then we write a message and restart the whole game, or quit out (q quits the game).

If, however, they typed 1 in, they're asked this level's multiplication question. If they get it right, it unlocks the current level and they can proceed, which we do by starting the `trainLoop` again but with only the tail of the levels (by using the `levelsAfterThisOne` variable).

All that remains is to show the definitions for `jumpingFutility` and `eatingFutility`. These actions are executed if the player tries to jump or eat.

All they do is print a message, then start the game at the very beginning by returning trainLoop applied to the initial levels.

```
jumpingFutility :: IO ()
jumpingFutility = do
  putStrLn "You try to jump out of the train."
  putStrLn "You fail and die."
  trainLoop levels

eatingFutility :: IO ()
eatingFutility = do
  putStrLn "You see a delicious looking cupcake."
  putStrLn "You eat it. It's a time travel cupcake!"
  trainLoop levels
```

18.5 Homework

Write a program that prints your birthday on the screen. Write another program that prints two numbers added on the screen, then one for multiplied, then subtracted.

19 Skwak the Squirrel

Games! We saw the Fridge game. It took place in one single "room" which made it very limited. Then, the Train game didn't really give you any freedom, but at least it had more rooms.

Next we'll see a game that lets us imagine that we're a squirrel and we live in a tree. Our new game will only have **two** areas, but it will provide more capability than before. And, in the process, we'll get some more practice with all the things we've seen so far.

If we wanted to make a game that is a tiny bit more like a real text adventure game, it'd have to let the player move around between its game areas.

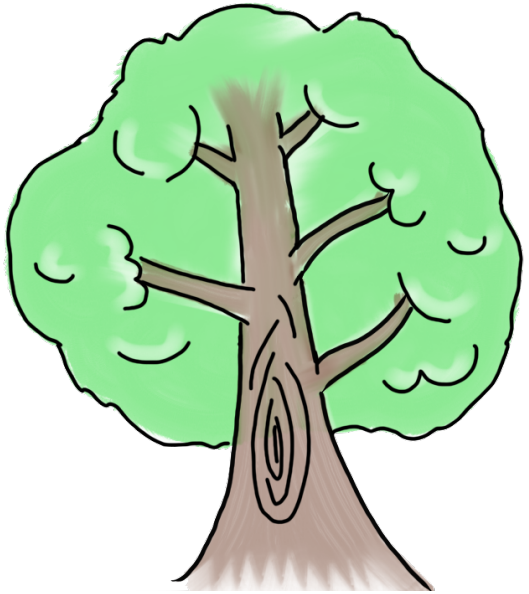
To keep it ultra-simple, let's say we (our squirrel) could be able to go between only the inside and outside of the tree, perhaps. The

program we'll be discovering in this section is a lot more complicated than the Fridge game, but it's about one of the most simple, basic text adventures possible.

Let's look at the types first:

```
data GameObject = Player
                | Acorn
  deriving (Eq, Show)
data Room =
  Room Description [GameObject]
    deriving (Show)
type Description = String
type Inventory = [GameObject]
type GameMap = [Room]
type GameState = (GameMap, Inventory)
```

We have a `GameObject` type whose values can be either `Player` or `Acorn`. Fairly straightforward, this is just a **sum type** like we've seen before. What about `deriving (Eq, Show)`, though? Well, this is a way to make a type become an instance of these typeclasses without having to write the code for it manually ourselves. Being an instance of `Eq` means we can use `(==)` and other comparison functions on values of this type, and being an instance of `Show` means it can be converted to strings with the `show` function.



Now, the `Room` type is a **product type** and it has a type constructor called `Room` as well as a value constructor of the same name, which is used to make values of the `Room` type. It has fields of `Description`, and a list of type `GameObject` which is used to hold the contents of the `Room`. So, because of the way the `GameObject` type is defined, a `Room` can have one or more `Player` or `Acorn` values in it. Our game will only ever have one of each in the whole game.

Next are a bunch of type synonyms which should be pretty easy to understand by now, possibly with the exception of `GameState` which is a 2-Tuple (otherwise known as a pair) of `GameMap`, and

`Inventory`, which is a list of `GameObject`. `GameMap` is a list of `Room`.

The `GameState` type will be what we use to store all the changing pieces of our game as the player plays it. The `GameMap` will be all the rooms in the game, and the `Inventory` is what the player is holding moment by moment.

Let's see a definiton for the initial state of the game and the `main` function, that will start the game.

```
initialState :: GameState
initialState =
  ( [ Room "You are inside a tree." [Player]
    , Room "You are outside of a tree." [Acorn]]
  , [] )

main :: IO ()
main = do
  putStrLn "Welcome to Skwak the Squirrel."
  putStrLn "You are a squirrel."
  gameLoop initialState
```

Now we see the `main` function, which as we know is the `IO` action that will be executed by Haskell when we compile and run our program.

All it does it announce the game, then runs a function called `gameLoop` using the `intialState` which is the state that the game should start with. The `gameLoop` contains the bulk of the program, and we'll see it in a moment.

First, Let's look at `initialState`. This is a `GameState`, which as we know from the types is a 2-Tuple that has a list of `Room` then a list of `GameObject`. We start our game with two rooms... we use the `Room` constructor to build the rooms. We have the outside and the inside of the tree. The `Player` is inside, and the `Acorn` is outside.

Let's look at the `gameLoop` function now:

```
gameLoop :: GameState -> IO ()
gameLoop (rooms, currentInv) = do
  let currentRoom =
        case findRoomWithPlayer rooms of
        Just r -> r
        Nothing -> error $ "Somehow the player "
                           ++ "ended up outside the map!"
      possibleCmds =
        validCommands currentRoom currentInv
  if playerWon (rooms, currentInv)
  then gameOverRestart
  else do
    describeWorld currentRoom currentInv possibleCmds
    takeActionThenLoop
      currentRoom currentInv possibleCmds rooms
```

This function takes a `GameState` and returns an `IO` action. It's the `main` functionality of the game. Each move the player makes goes through the game loop once, which is why it's called a loop, because it's like a circle.

We see that we can have a `let` expression in our `do` block which essentially sets up temporary variables in the `do` block from that point onwards.

We're finding and grabbing the current room from the list of rooms, or throwing up our hands if it can't find the `Player`.

Then we work out which commands are valid for the current room and the current inventory. Some player commands are only possible with some combination of `Acorn` being in the room with the player, or in the inventory.

Once that is done, we check if the player has won with an `if` expression, and if so, we tell the player they've won and offer to play again. If they didn't win, then we describe the world to them at the room they're in, and let them take an action by grabbing their command, then go back again to the start (using the `takeActionThenLoop` function).

Now we'll go through the remaining functions. Note, though, that we can use any expressions (such as `if`) with no trouble in a `do` block for `IO`, even other nested `do` blocks, so long as those expressions result in an `IO` value of some kind. We'll see in a moment that all of those functions we just discussed yield `IO` values of some kind.

```
findRoomWithPlayer :: [Room] -> Maybe Room
findRoomWithPlayer rooms =
  L.find (\(Room _ obs) ->
            any (== Player) obs)
         rooms
```

We can tell by the name that `findRoomWithPlayer` should return the `Room` whose objects include the `Player`. We're returning a `Maybe Room` because it's technically possible that the `Player` might not be in any `Room`. If that's the case, there's a problem, because the game won't work. That's why there's an application of the `error` function in `gameLoop` if `findRoomWithPlayer` can't find a `Player`.

We're using a lambda as our Room-testing function here. The way find works is it looks through the list passed into it, applies the predicate section function `(== Player)` to each item, and if it finds one that returns `True`, it returns that one, wrapped in the `Just` data constructor from the `Maybe` type. If it doesn't, it returns the `Nothing` value (also from the `Maybe` type.

In depth, our lambda takes a `Room`, pulls it apart using pattern-matching to get at the objects (which is a list of `GameObject`), names that `obs`, and then passes that to `any`, which will check to see if any of them return `True` for the function `(== Player)` which checks to see if something is equal to the `Player` value.

Next we'll see the function that crafts the valid commands for a particular room and inventory combination:

```
validCommands :: Room -> Inventory -> [String]
validCommands (Room _ gameObjs) invItems =
    ["go"] ++ takeCommandList
    ++ dropCommandList ++ ["quit"]
  where
    takeCommandList =
      if somethingToTake gameObjs
      then ["take"]
      else []
    dropCommandList =
      if length invItems > 0
      then ["put"]
      else []
```

This function takes a `Room` and an `Inventory` and returns a list of `String` values that are all the commands that a player can type in depending on the current data: whether there is anything to take (in the room), or drop (from inventory).

It's using the list concatenation operator `(++)` to join the `String` lists together into one list to return, and two definitions with `if` expressions to ensure we only put "take" or "put" on the command list if it's appropriate.

```
somethingToTake :: [GameObject] -> Bool
somethingToTake objs =
  any (/= Player) objs
```

Of course, we can **eta-reduce** this function as we know by now (that is, get rid of the repeated trailing arguments):

```
somethingToTake :: [GameObject] -> Bool
somethingToTake = any (/= Player)
```

Here we take a list of `GameObject` which is used by `validCommands` as the list of objects in the room that can be taken. We're using it to work out if there is anything that is not a `Player` in the list. The `any` function takes a predicate function, evaluates it on each item of a list, and returns `True` if any of the evaluations return `True`, otherwise `False`. The function `(/=)` means not equal to, and comes from the `Eq` typeclass. The end effect answers the question "Are there any non-player objects in this room?".

As we've seen before, the use of parentheses around `(/= Player)` makes it into what's called a **section**: it creates a function of one argument, having applied one of the 2-arguments required by the operator. Here we're creating a function that takes a `GameObject` and returns a `Bool` that indicates whether the `GameObject` was `Player`.

```
playerWon :: GameState -> Bool
playerWon (rooms, currentInv) =
    any hasAcornAndInside rooms
  where hasAcornAndInside (Room desc objs) =
          desc == "You are inside a tree."
          && any (==Acorn) objs
```

This function answers the question "Has the player won yet?" The player has won if the acorn is in the tree (and not in the player's inventory). Pay careful attention to the indentation (where the lines start). This matters!

We test for this by pulling the `GameState` tuple apart into a variable each for rooms and current inventory. We then go through all the rooms, and using the `any` function again, check if there's one which has the description "You are inside a tree." that also has the Acorn in its objects list.

Note the use of the `&&` operator, which takes two `Bool} expressions and returns \{True` if they're both `True`. This is called the **logical and** operator. It's one way we can connect up logic expressions into larger chunks of meaning. There is also an `(||)` operator that is **logical or**, which will return `True` if either of its inputs evaluate to `True`, and a `not` function that takes only one `Bool` expression and returns the **logical inverse** of it (that is, if it's `True`, it returns `False` and vice-versa).

Our `playerWon` function is a not the best way we could write it, because we're relying on the description of the room to check if it's the inside room. This gives us a lot of room for errors to creep in by accident. We did this to keep the types a bit simpler as you're learning. It's bad because what if we changed the description in the actual `Room` data, but then forgot to change this check `String`. It would mean it'd be impossible to win the game.

A better way would be to pull this description into its own definition that is defined in only one place. The best way, though, would be to have an identifier in the `Room` data type representing the type of `Room` it is, and to check against this to see if this `Room` was the winning goal `Room`.

```
gameOverRestart :: IO ()
gameOverRestart = do
  putStrLn $ "You won!"
    ++ "You have successfully stored the acorn"
    ++ " for winter. Well done!"
  putStrLn "Do you want to play again? y = yes"
  playAgain <- getLine
  if playAgain == "y"
  then gameLoop initialState
  else putStrLn "Thanks for playing!"
```

Our function to check if the game is now over uses a `do` block to print some messages congratulating the player, then asks if they want to play again, collects a line of input from them, and restarts the game from the beginning by using `initialState` if their input equals `"y"`. Very straight-forward!

```
getCommand :: IO String
getCommand = do
  putStrLn "What do you want to do?"
  getLine
```

`getCommand` is a simple little action that prints a query to the player, gets that response and returns it. The type is `IO String`, which is the same type as `getLine`. Notice that as `getLine` is the last line of the function, we don't need to use anything to pull the value out here, because it has the return type already. We'll see how it's used now in `takeActionThenLoop`:

```
takeActionThenLoop :: Room ->
                      Inventory ->
                      [String] ->
```

```
                    [Room] ->
                    IO ()
takeActionThenLoop currentRoom
                   currentInv
                   possibleCmds
                   rooms =
  do
    command <- getCommand
    if any (==command) possibleCmds
    then case command of
      "go" ->
          do
            putStrLn "You go..."
            gameLoop $ movePlayer (rooms, currentInv)
      "take" ->
          do
            putStrLn "You take the acorn..."
            gameLoop $
              moveAcornToInventory (rooms, currentInv)
      "put" ->
          do
            putStrLn "You put the acorn down..."
            gameLoop $
              moveAcornFromInventory (rooms, currentInv)
      "quit" ->
          putStrLn $ "You decide to give up."
                   ++ " Thanks for playing."
      _ ->
          do
            putStrLn "That is not a command."
            gameLoop (rooms, currentInv)
    else do
      putStrLn $ "Command not possible here,"
               ++ " or that is not a command."
      gameLoop (rooms, currentInv)
```

Even though this is one of the largest functions in the program, it's reasonably simple. It's structured by firstly taking a command from the user, then checking if the command is one of the valid ones for the player right now. If not it complains that it can't understand and just goes back to the `gameLoop` with the current data. However, if it does actually match against a valid command, it runs through the case expression trying to find a match.

Each of the commands has a corresponding function which results in a modified `GameState`, which is then passed back into the `gameLoop` function for a further turn. We'll see each of these functions momentarily.

Notice that we use the `($)` function all over the place, such as to apply `gameLoop` to the result of the evaluation of the command functions, such as `movePlayer`.

Also notice that there is a pattern match for "`_`", which for completeness will match on any other values for the `command`. In our case there aren't actually any other commands, but not having this is a potential source of problems in the future as things change, and Haskell will complain if we leave it off. Take a moment and think what would happen if you added a new command to the `validCommands` function, but didn't add it to the `takeActionThenLoop` function. If we keep this underscore catch-all, then play the game, the game will tell us that our new command is not recognised as a command. If we leave it off, though, our program will crash. That's why it's better to have **total functions**.

These next three functions are to adjust the game state when the

player does something.

```
movePlayer :: GameState -> GameState
movePlayer (rooms, inv) =
    (newRooms, inv)
  where
    newRooms =
      map adjustRooms rooms
    adjustRooms (Room d objs) =
      if any (==Player) objs
      then (Room d (filter (/=Player) objs))
      else (Room d (Player : objs))

moveAcornToInventory :: GameState -> GameState
moveAcornToInventory (rooms, inv) =
    (newRooms, newInv)
  where
    newInv =
      Acorn : inv
    newRooms =
      map adjustRooms rooms
    adjustRooms (Room d objs) =
      Room d (filter (/=Acorn) objs)

moveAcornFromInventory :: GameState -> GameState
moveAcornFromInventory (rooms, inv) =
    (newRooms, newInv)
  where
    newInv =
      filter (/=Acorn) inv
    newRooms =
      map adjustRooms rooms
    adjustRooms (Room d objs) =
      if any (==Player) objs
      then Room d (Acorn : objs)
      else Room d objs
```

Our game only has two rooms, so the `movePlayer` function just goes over "all" two rooms using the `map` function, and applies a function that takes the `Player` out if it's there using `filter`, or puts it in if it's not there, using the list-prepend function `(:)`. It does this by pattern-matching the `Room` into its pieces, then reconstructing it with a new `GameObject` list.

The `moveAcornToInventory` function returns a new 2-tuple (a `GameState`) that is comprised of an adjusted set of rooms by mapping a deconstructing-reconstructing function that uses `filter` to remove all acorns from the rooms' `GameObject` list (there should only be one anyway), and an adjusted inventory by prepending an `Acorn` to it using `(:)`. This "moves" the `Acorn` into inventory.

The `moveAcornFromInventory` does the reverse of this. The mapping function that puts the `Acorn` into the room where the player is is quite interesting because it has an `if` expression to detect if a particular room's game objects contains the `Player`, and if so, it prepends an `Acorn`, otherwise it just re-builds the passed in room as it was before.

```
describeWorld :: Room ->
                 Inventory ->
                 [String] ->
                 IO ()
describeWorld currentRoom
              currentInv
              possibleCmds =
  do
    putStrLn $ describeRoom currentRoom
    putStrLn $ describeInventory currentInv
    putStrLn $ describeCommands possibleCmds
```

This is pretty straightforward, it just uses other functions to describe to the player which room they're in, what they're carrying and also which commands they can use here.

Notice, again, that we're using another `do` block to make a single `IO` action out of a bunch of `putStrLn` function applications.

19.1 More on the ($) Function

Also we see the function `($)` again at work, applying the function on the left of it to the result of evaluating everything on the right of it. The function application operator is often used as above, to avoid having to type brackets around expressions, as you probably know by now. It has the lowest possible precedence, which means it's evaluated last, after all the other parts of an expression.

The type signature of this is `($) :: (a -> b) -> a -> b` which means it takes a function from `a` to `b` (on the left of it) and an `a` (on the right of it), and applies the function to the "a" which gives a `b`. Obviously this is an extremely general function because `($)` can apply to any function, and any value as long as they fit together.

Again, it's important to realise that even though the type signature of `($)` says it takes a function from `a` types to `b` types, it doesn't **have** to be different types, but they **can be** if you like. So, `(+3)` fits the description even though its type is `Num a => a -> a` and this is because `(a -> b)` goes from one "any type" to another "any type", so **not necessarily** the same type. `(a -> b)` is actually the type that fits any type of function at all!

There is no difference in result between writing `(+3) $ 7` and writing `(+3) 7`, but when you start adding more arguments it starts to matter, like in `(+3) $ 7 * 5` which gives `38`, a different answer than `(+3) 7 * 5`, which is 50. From this we can see that `($)` changes the order that things are evaluated in, just like putting parentheses around some things (so, `(+3) $ 7 * 5` is equivalent to `(+3) (7 * 5)`).

19.2 Mid-Lesson Homework

Your homework (yes, mid-lesson homework) is to experiment with the `($)` function, and explicit bracketing, and explore the differences and similarities if you can.

19.3 Continuing On

Next, the three functions that `describeWorld` uses:

```
describeRoom :: Room -> String
describeRoom (Room desc objs) =
  desc ++ if any (==Acorn) objs
          then " There is an acorn here"
          else ""

describeInventory :: Inventory -> String
describeInventory [] =
```

```
    "You are holding nothing"
describeInventory inv =
  "You are holding: " ++
  (concat $ map show inv)

describeCommands :: [String] -> String
describeCommands commands =
  "Commands: "
  ++ (L.intercalate ", " commands)
```

The `describeRoom` function takes a single `Room` and pattern-matches it into the `desc` and `objs` variables, which are the description of the `Room`, and the objects that the `Room` contains respectively. Notice that we're using `(++)` to append the result of an `if` expression to the `Room`'s description here. The `if` expression uses a **section**. Remember that a section is effectively a partially applied operator. In this case, `(==Acorn) :: GameObject -> Bool` will check if something is an `Acorn`, and return a `Boolean` value for that (`True`/`False`).

Let's think some more about the function `any :: Foldable t => (a -> Bool) -> t a -> Bool` for a moment. This is a function that takes a predicate from types `a` to `Bool`, a `Foldable t a` and then returns a `Bool`. This function will tell us if any item in the list "satisfies" the predicate. That is, if any item in the list returns `True` for our predicate function.

Or, partially applied to the `(==Acorn)` section as in `any (==Acorn)`, it will tell us if there's an `Acorn` in a supplied list! If there **is** an `Acorn`, it'll say so. If not, just an empty `String`.

Next we'll look at the `describeInventory` function. There are two definitions on this function. The first is for when there is nothing in the passed-in `Inventory`, so we're matching on an empty list. This is simple.

The second definition is where the meat is. If the match with the empty list failed, that means there must be something in the `Inventory`, which we now pattern-match to the "inv" variable.

We then use the list concatenation operator (++) to append the result of an expression to the end of the beginning of a description of what they player is holding. Of course, this works because `String` is `[Char]`, so therefore a list.

The expression we're appending is `(concat $ map show inv)`. We can break this apart into its pieces. Firstly, we now know about the `($)` function, and can think about it like it's bracketing like this: `(concat (map show inv))`.

Thinking about the expression `map show inv`, we can see that this takes the `inv` list and returns a new list that it builds by applying the `show` function to each `GameObject` in it, turning it into a list of strings. We can see this clearly if we check its type: `map show :: Show a => [a] -> [String]`.

The function `concat :: Foldable t -> t [a] -> [a]` takes a `Foldable`-wrapped list of type `a`, and returns a list of type `a`. So in this case, because we're passing it a `[String]`, which as you know is the same as a `[[Char]]`. So that means the `Foldable t` that `concat` needs will be the list type, so a list

of list of `Char`. The `concat` function will concatenate (join) that `[String]` to be a single `String` (or turn the `[[Char]]` into a `[Char]`, same thing), by joining all the strings into one `String`.

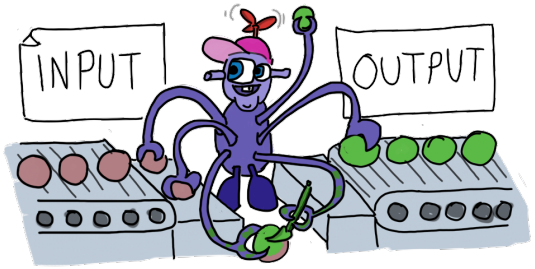
The end result is that `describeInventory` will either tell the player they're holding nothing, or that they're holding a list of what they're holding. The fact that there will only ever be one item in the list means we don't need to comma-separate it or anything.

Next is the `describeCommands` function. It takes a list of commands, which are just `String` values, and calls the `intercalate` function from `Data.List` on that it. This is a pretty neat function. Its type is `[a] -> [[a]] -> [a]`. The first argument is a separator, and the second is a list of items of the same type. It then joins them all together into one big list after putting the separator between each of the items. So, here we're using it to separate all the commands by comma-space so that it reads nicely.

20 Basic Input

Covers: basic input, simple `do` blocks, simple function application, simple definitions.

We're going to teach you just enough IO that you can write beginning programs. More will come later as we need it to do more capable programs.



We'll begin straight away on the guided program building.

20.1 Guided Program 1

Task: Make a program to ask for a name, then say hello to that person using their name.

Taking this problem apart, there seem to be three pieces. Let's begin with the piece we can do easily: ask for a name. This is simply

printing a string on the screen, which we know how to do:

```
main :: IO ()
main = putStrLn "What is your name?"
```

So we've solved a third of the problem. Next, we need to actually get the name of the user. If we think of the functions we know about, the `getLine` function is the obvious contender here, which has the type `IO String`. We're going to have to compose this with our `putStrLn` action. By now, handily, we know about `do` syntax which lets us combine multiple `IO actions` into one.

You can think of each expression in an `IO do` block as a piece of the composed `IO` action, which is exactly what it is. You can write any kind of expressions you like in that `do` block, so long as it will evaluate to `IO ()`.

As we've seen, there are two exceptions to this rule, and they are `do`-block let notation, which is used for definitions of pure expressions, and the `<- syntax`, which is used for defining variables as the "inner values" of `IO` actions that we want to connect to some other part of the rest of the code in our `do` block.

So, right away, we notice `getLine`'s type isn't what we need it to be. It's `IO String`, not `IO ()`. However, we know by now that we can use `<-` to "get" a pure value out of an `IO` action so long as the code is still in `IO`. This `do` block is an `IO` action, so we can do it here. Let's add the `do` block and then use `getLine` to define a variable called `theirName` as the `String` value it pulls out. Remember, though, that the last expression in a `do` block must be of type `IO ()`, so for now we'll use a new function that we haven't seen called `return` to create a value of this type. We'll use the expression `return ()` to put `()` into `IO` and make **that** the end expression of our entire `do` block:

```
main :: IO ()
main = do
  putStrLn "What is your name?"
  theirName <- getLine
  return ()
```

The `return` function places a value into an `IO` action. It doesn't have anything to do with returning values **from** actions, just with putting values **into** actions. It's possibly a bad name to have for your understanding at the beginning, so we apologise about that.

The last piece is to print their name out with hello in front. To do this, we'll need to use the `putStrLn` function again, and also the `(++)` operator that joins (concatenates) two lists into one. `String` is simply a list of `Char` as we know, so this will work fine.

Choose your variable names wisely! Good naming is one of the most difficult tasks in programming. It can help or hinder future readers of your code, including yourself. Names should always help to explain your code as much as possible.

Notice in the next piece of code that we don't need to have the `return ()` function application now because of `putStrLn`. However, we are using brackets around the application of `(++)` to its arguments because if we didn't, `putStrLn` would just take the one

`String` as its argument, and `(++)` would be being applied to an `IO ()` value on the left side rather than a `String`.

```
main :: IO ()
main = do
  putStrLn "What is your name?"
  theirName <- getLine
  putStrLn ("Hello, " ++ theirName)
```

This program is just about the simplest possible program that we could write that uses both input and output in Haskell. If you'd like to, try playing with it a little, by changing to different `String` values. Don't get discouraged if things go wrong.

20.2 Guided Program 2

Task: Write a program that asks you for your name, then your pet's name, then tells you the info back.

This is still a very simple program, so we'll just use what we know about `putStrLn`, `do` blocks, `(++)` and `getLine` to build our program:

```
main :: IO ()
main = do
  putStrLn "What is your name?"
  theirName <- getLine
  putStrLn "What's your pet's name?"
  petName <- getLine
  putStrLn ("Your name is "
            ++ theirName
            ++ " and your pet's name is "
            ++ petName)
```

As we've said before, it's very important to get your indentation correct in Haskell. If you don't, Haskell won't know what you mean, and will often complain with strange sounding errors. The indenting is good because it stops us from having to use a lot of bracketing, because Haskell can use the indenting to work out what you mean.

You should experiment and try out different indentations when you have a program that compiles to see what works and what doesn't, and to get familiar with some of the errors that appear when things go wrong.

The above program is almost the same as the first one, except this time we're extracting **two** variables from the user using `getLine`. Notice that you can chain operators if they take two of the same type of arguments (such as `(++)`) and then the result of the first application will be fed in as the first argument to the second operator, as you would expect it to work.

Let's look at a different way to write a similar program, which may surprise you a little.

20.3 Guided Program 3

Task: Write a program to ask for the time, then to write "again!" and then ask the user the time again and print out both times.

It's virtually the same idea for thinking through how to build this program, however we're going to create a separate action to be re-used in our main action.

Any time you would end up repeating yourself, it's a great idea to re-use a fragment or value. There's no reason not to separate it out by making a definition for it. Then you don't have to repeat yourself! This is one of the core reasons to write computer programs. You should strive to get the computer to do as much of the repetitious work as possible.

```
whatTimeIsIt :: IO String
whatTimeIsIt = do
  putStrLn "What time is it now?"
  getLine

main :: IO ()
main = do
  timeString <- whatTimeIsIt
  putStrLn "Again!"
  timeString2 <- whatTimeIsIt
  putStrLn ("Ok, you said it was "
            ++ timeString
            ++ " and then you said it was "
            ++ timeString2)
```

So `whatTimeIsIt` is an `IO String` action. It has the same type as `getLine`, so we can treat it exactly the same. Looking at it, you can see `getLine` is the last value in its `do` block, so it's just using a `do` block to connect outputting a question up with asking the answer, and returning that string (in `IO`, of course).

Here we see the difference between using action results and pure expressions very clearly. We can use the action twice to get two potentially very different values out. We do so using `<-` again, and set two different variables to their answer.

Then, we just output the result sentence. All of this is joined up using a do block as before. Nice!

20.4 A Little More About IO

All an `IO` action is, is a description of how to enact some execution of code later on. The `<-` syntax doesn't actually "get" anything out of anything else. You can think of it like it does, but what it really does is tells Haskell **what to do** with action values to combine them. It tells Haskell how to connect up the producing-part of a piece of code that queries the user for input to the rest of the consuming-part of the program, for example.

When we write `do` blocks, all we're really doing is telling Haskell what relationship the smaller pieces should be in compared to each other, and then when the program is executed, things happen in the right order. This will become clearer with more practice, and in later volumes.

20.5 Your Turn

Now it's your turn. When you do these exercises, do them without looking at the supporting explanation documents in this lesson. If you have to look, you can, but mark the exercises you had to look for, and re-do them again after you've finished doing them all. You should do them again the next day and so on trying without looking until you can easily do them without any problem at all. Make up some programs of your own that involve asking questions and replying with the inputted data. Don't try to do anything more complicated than this, yet. You can do this. Let's go.

20.6 Reader Exercise 1

Task: Write a program to print a welcome on the screen.

20.7 Reader Exercise 2

Task: Write a program to ask for someone's last name, then print it out on the screen.

20.8 Reader Exercise 3

Task: Write a program that asks two personal questions about the user, and tells their responses back.

20.9 Reader Exercise 4

Task: Write a program that says it's going to ask the user to pick three numbers, then uses a separate `IO String` action that prompts a user to pick a number, then does it again two more times, then tells them the numbers they picked (note we're not actually using numbers here, just `String` input and output).

20.10 Reader Exercise 5

Task: Write a program to ask the user where they live, then write a message saying something telling them you've heard that's a great place to live, using the place name in a sentence.

21 Getting Set Up

Thankfully, these days, Haskell is a reasonably easy programming language to get started with. Having said this, **all** programming languages aren't particularly easy at the beginning.

We have two ways that we recommend to get started.

21.1 Mac Set Up

If you have a Mac, and a small amount of money to spend, then we very much recommend checking out `http://haskellformac.com`. It's a really simple, easy way to get off the ground in Haskell, and complements this book quite well. The publishers of this tool have made it so you can pretty much just dowload it and start typing Haskell in and get compiling without worrying too much about the tooling normally involved.

Beautiful!

21.2 Manual Set Up

The alternative to this is that you need to know how to use the command line at least enough to install software. Again, thankfully, setting up Haskell here is a lot easier than it has been very recently in the past. If you don't know how to use the command line, though, there's a nice little tutorial here: `https://www.davidbaumgold.com/tutorials/command-line/`.

To get set up with Haskell in this way, go to `http://www.haskellstack.org` and follow the instructions. The people who set up the **stack** installer have spent quite a bit of effort on making the install process reasonably easy.

21.3 Questions & Community

Haskell has a very helpful and enthusiastic community. If you go to `https://www.haskell.org/community` you can read about the latest ways to connect with the community and get help if you have any problems with getting started, or any other questions at all.

22 Frequently Asked Questions

Many of our readers ask these questions, so this chapter addresses them for when other readers have these same questions.

22.1 Volume 2 and Language Features

Question: Are you planning **Volume 2** of the book? Or another book with advanced topics (such as Monads, Combinators, the Y combinator, beta-reduction, Generalized Algebraic Data Types, etc.)

Answer: Yes, we're definitely planning Volume 2. It's currently being written and will be released in beta book soon.

We like to draw the teaching of topics out into two or three phases, depending on how you see it. These are: 1. getting the student used to reading/seeing a language feature in action in many places, but not talking about the general case too much 2. getting the student to use a language feature in a few places until they're familiar with using it on their own and know when it should be used and only then 3. understanding the general case, and naming the feature.

Sometimes we will introduce the third "phase" with the first, sometimes with the second, and sometimes only afterward. It really depends on the topic, how complex it is, and how easy it is to understand.

Having said this, Volume 1 has seen Monad and Combinators the entire book. You might be talking about the combinator pattern here. Again, our **whole book** uses this approach of building up small functions that do tiny things, and then building our programs out of these pieces. That's the combinator pattern.

What about beta-reductions? Well, that's just function application, so yes, we've been doing that the entire book as well.

However, some of these topics, such as the Y-combinator and Generalised Algebraic Datatypes are incredibly advanced. While we do plan more volumes than just 1 and 2, it's worth noting that our books exist to describe things to people so they can use them. It's no good studing GADT's if you aren't extremely well versed at **using** normal ADT's first, and even then it's only useful if you're building something that is worth the extra complexity.

The Y combinator is an example of something so advanced that most people who understand it have trouble explaining how to apply it to a real use-case, or what it's for. Its sole purpose is to allow you to do calculation in a functional language without any named variables or functions. That is, to show how you can "bootstrap" computation by having nothing more than function application.

In terms of **naming language features** when we use them, we often find people get **incredibly confused** when you throw a multitude of jargon at them. Haskell has a **lot** of names for things, and we've witnessed many times where students have started to understand something, then exclaimed the equivalent of "Oh, **THAT**'s all ((feature)) is, I totally know how **THAT** works!" We'd rather you got an **understanding** of what you're doing so that when we explain what

things are, you have a practical base for them.

As a good example of this, notice how we don't use the name **Monad** at all in our book, and yet we use them the whole way through (with do syntax mostly. Or, notice how we don't use the word **currying** until about Chapter 13, but we use the **feature** the whole way through the book.

If we were writing a book for people who had used other languages before, we would definitely introduce more names earlier on, but even then the names can sometimes get in the way. As a good example, the proper name for **folding** in Haskell is **catamorphism**. It doesn't help you to know that when you start, and it's not even **that** helpful later on until you get to see how it contrasts to **anamorphisms**, which are things that **unfold**. What does that mean? It's when you build up a collection of things from a single seed value.

23 Many Thanks

We'd like to give you a heartfelt **thank you** for purchasing and reading this book. You've helped to improve learning Haskell for people who prefer this style of learning by supporting this book.

We'd love it if you shared this with as many people as you can, and asked them to purchase a copy if they are able to as well. Not only will they get ongoing updates for free, but it will also support the ongoing creation of more excellent learning material for beginners and beyond.

It's our plan to continue to write more volumes until we've done enough examples and exercises to take you through a broad understanding of how to read and write Haskell code to an intermediate level and further, where you can hopefully build whatever you'd like to, or at least have the confidence and knowledge to learn what you need to in order to do so.

This book and series is an **ongoing work** which means it will continue to be tweaked and adjusted as time passes, and so if anything sucked, was awesome, you got confused, or was really clear and you loved it, **please do** let us know on the feedback link on `http://www.happylearnhaskelltutorial.com` we'd **love** to hear from you! There are no stupid questions, so don't be afraid, you'll probably end up helping improve the quality of this book.

We hope you've had as much of an enjoyable time reading and using this volume as we have making it, and we'd love for you to

join us on further volumes.

— **GetContented**