# Happy Learn Haskell Tutorial Vol 1

http://www.happylearnhaskelltutorial.com/

***Written and Illustrated by*** *GetContented*

*(*enquiries@getcontented.com.au*)*

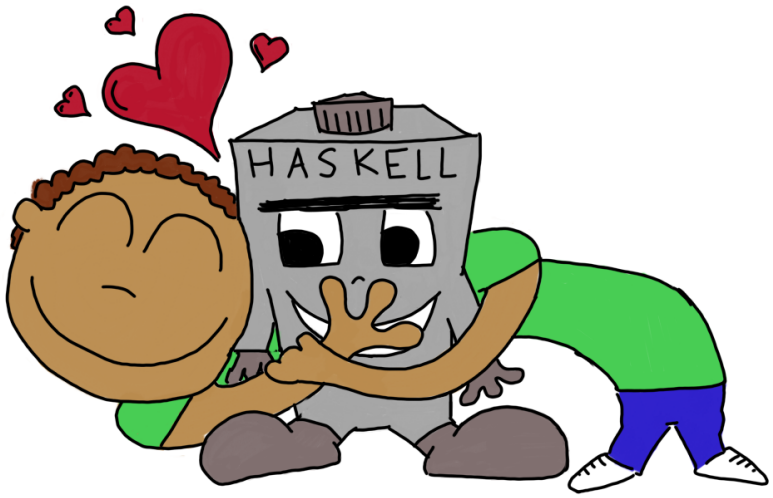
Updated on: August 14, 2021.

# 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.

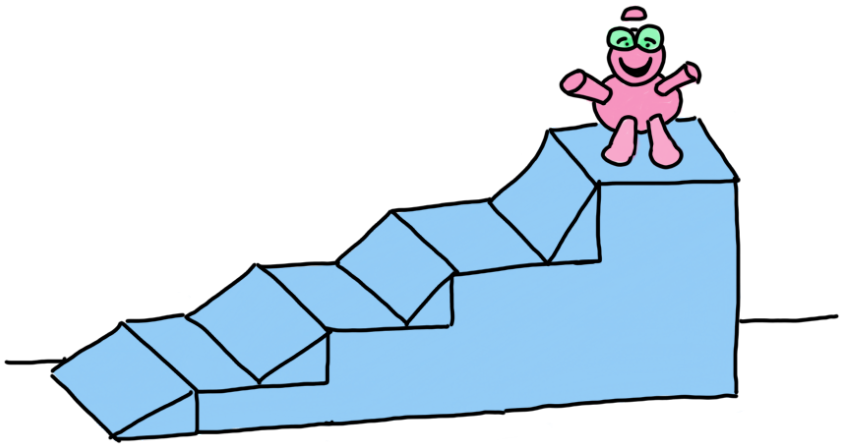
## 1.10 Stages Build Skill

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.



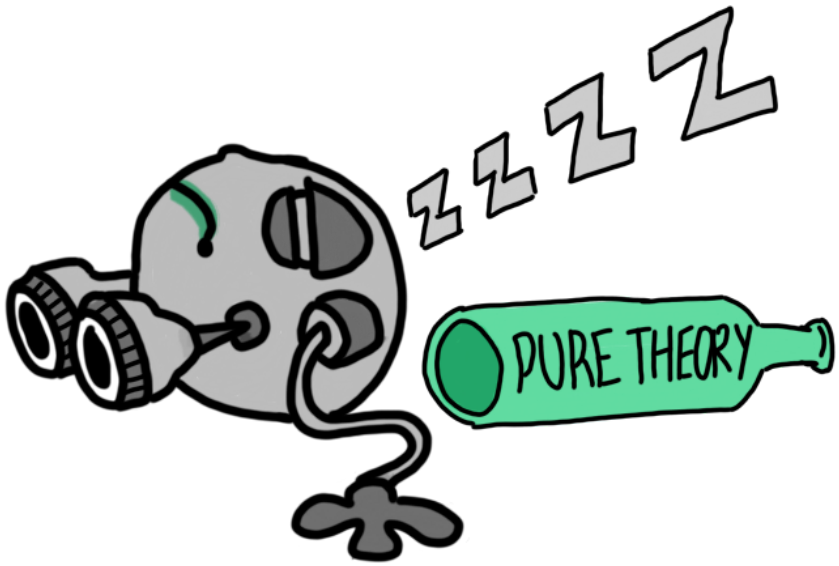
Many books don't address the subject of how to craft solutions, or they just leave you with nothing but some exercises and your own intuition. Most are focussed heavily on programming language topics first, and how you use them to do things second, if at all. You'll notice we're primarily interested 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.

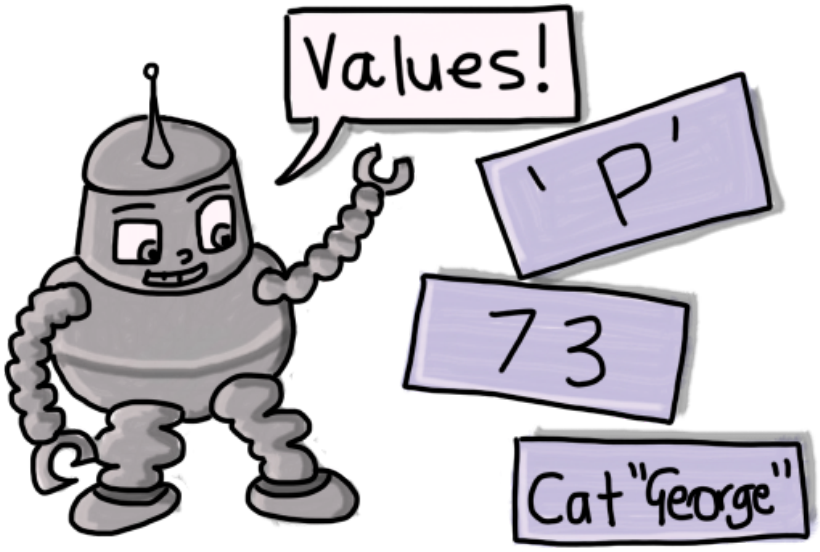
It's important to begin simply, because doing small steps helps you to avoid frustration and increases happiness!

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.

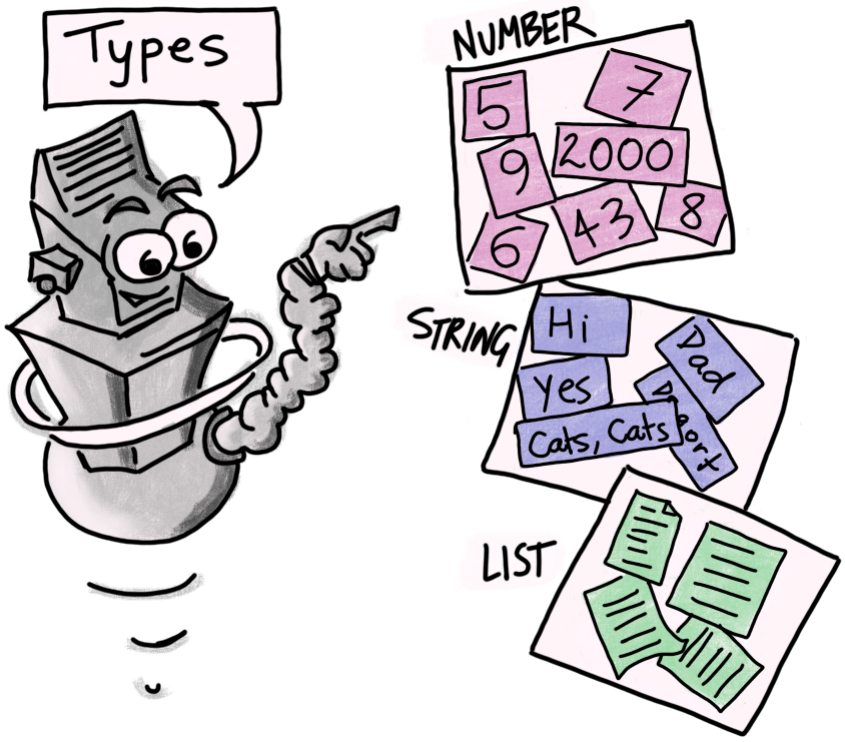
## 2.1 Values

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.

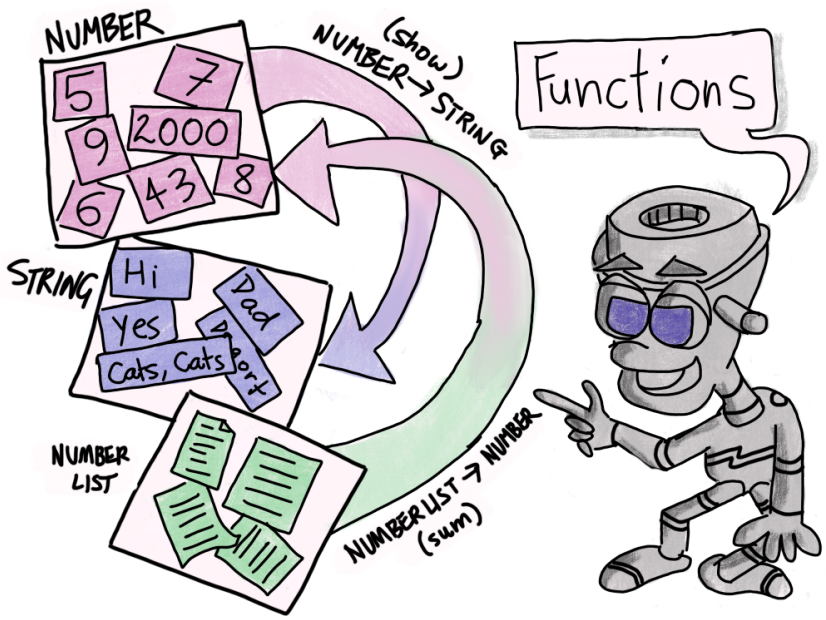
The `=` symbol relates (or we can say **binds**) the name or pattern on the left of it with the expression on the right of it. For example, `five = 5` tells Haskell that the name `five` means the value `5`.

## 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!



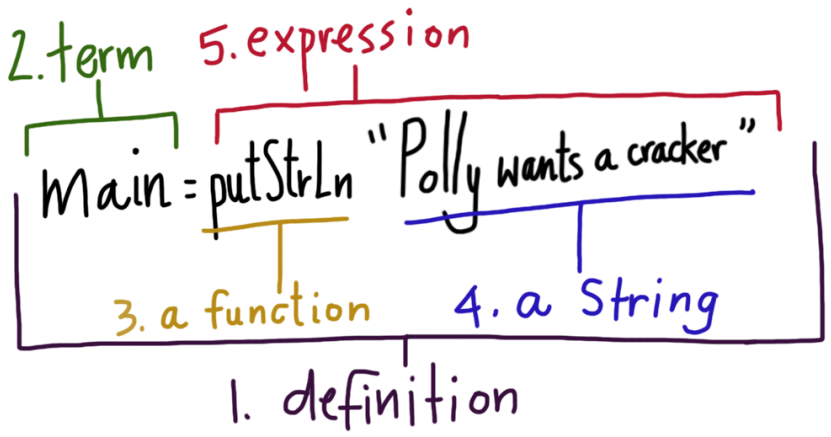

## 2.5 Our First Program

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

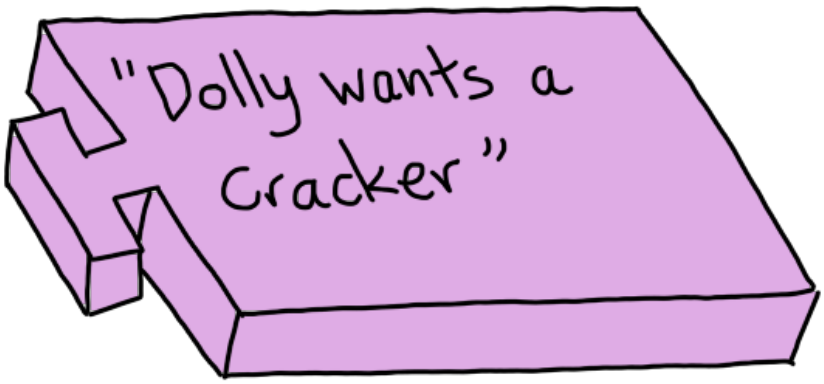
Here is a `String` value:

```
"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:

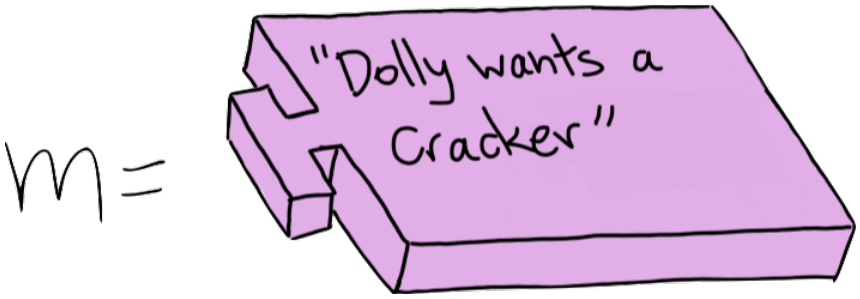


## 3.3 Definitions again

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

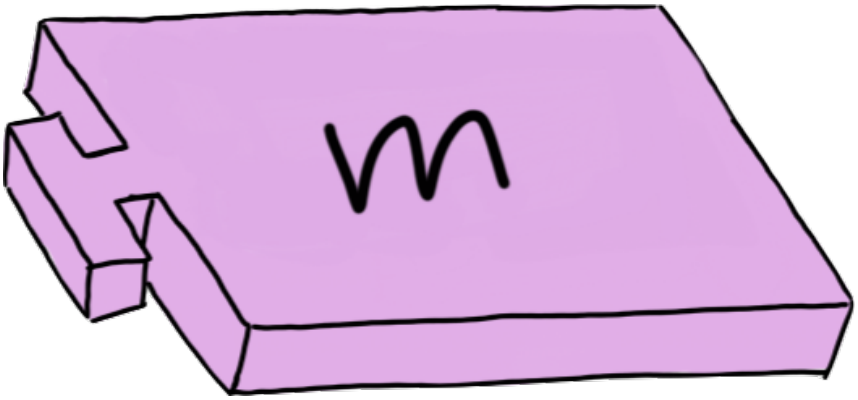
Now we'll see what it looks like when we include the code to describe the **type** of `m`, too.

```
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 ()
```

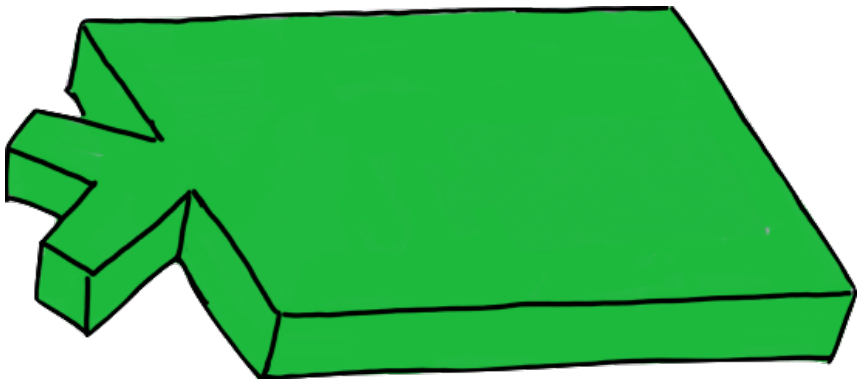
(**Note:** You will never need to write this type in your own code because it's already defined. We've just shown it here so you can see what it is, and how to work with it.)

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 func-

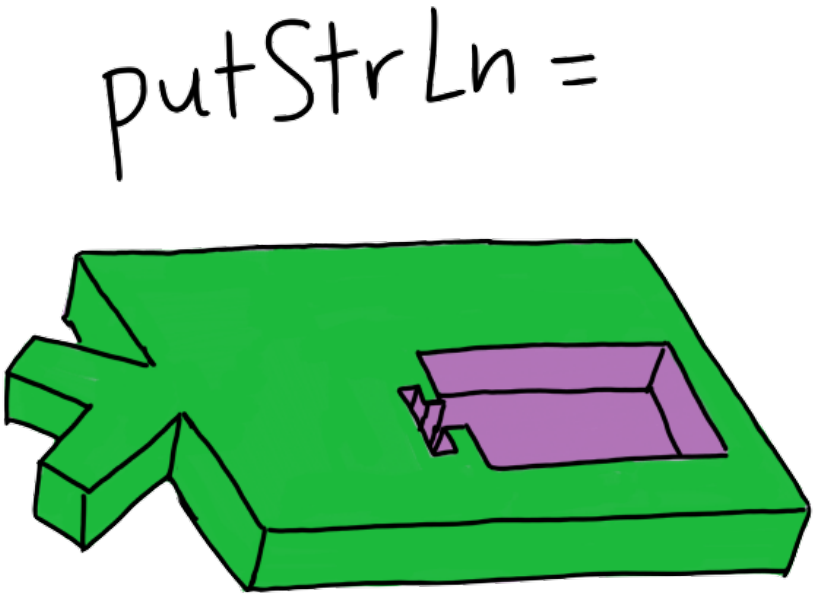
tion's input type, `(->)` signifies that it's a function, and goes between the input and output types, and `IO ()` is the function's output type.

## 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):



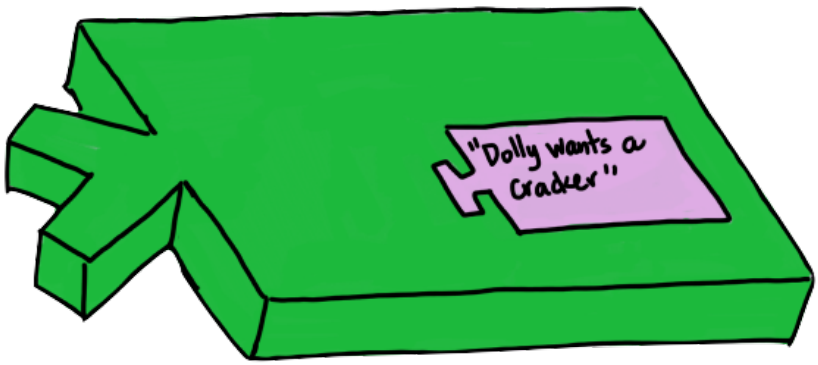
What about `putStrLn`? Well, it'd need be a value of type `IO ()` once it had a `String` popped into it. We could imagine that it looked like this, roughly:




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 ()`.



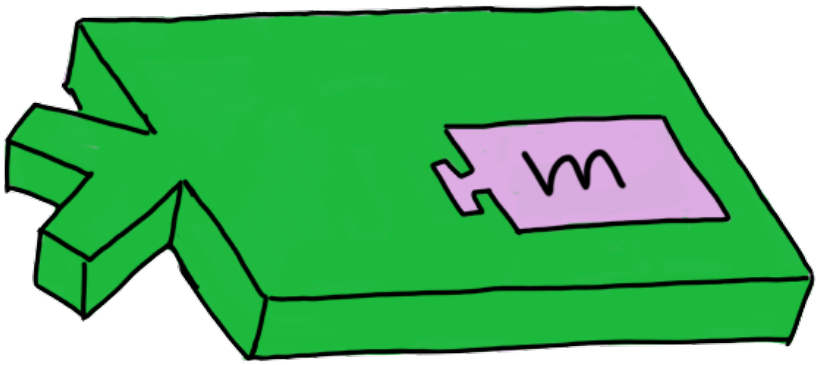
```
putStrLn "Dolly wants a cracker" :: IO ()
```

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

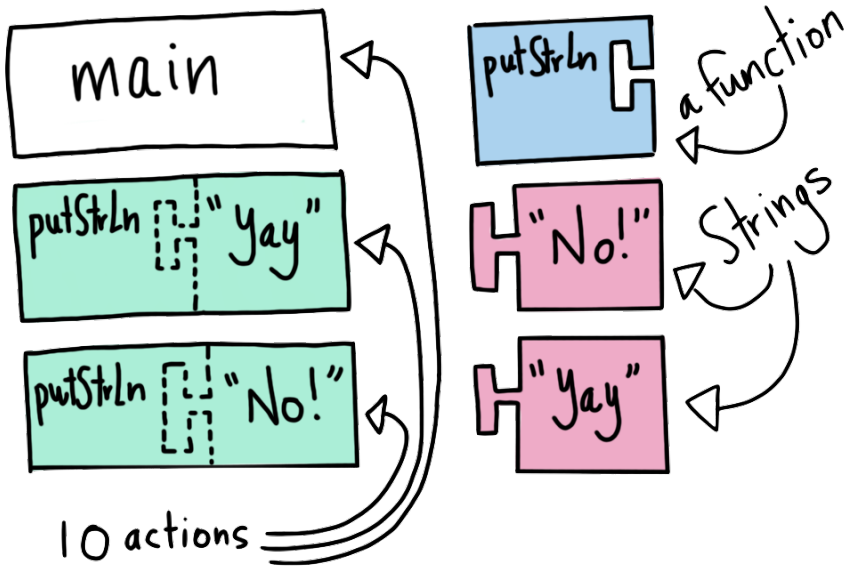
So this expression is an `IO ()` value, and because `main` needs to be an `IO ()` as well, we can see that one possible definition of `main` could be this `putStrLn` expression. Here's the whole program to print out the `String`:

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

Almost always when we write Haskell programs, we try to make sure our functions are **complete**. This means that we've written them in such a way that they work with **every possible value** for their input types.

## 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`). You can see how this means that functions are a mapping from **any** value of their input type to a particular value of their output type.

## 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 to it, 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 call this "Nothing of Interest" value **Unit**, and it's written like this: `()`. That is both its type, and how we write its value. Later you'll see that this is called an **empty tuple**, which is just another name for an emptly wrapper that can have nothing in it. 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. However Haskell won't let you confuse it by naming two different things the same identifier, so trying to is an

error.

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 seen this before: we just pulled the No! `String` out into its own definition (naming it with the identifier `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.

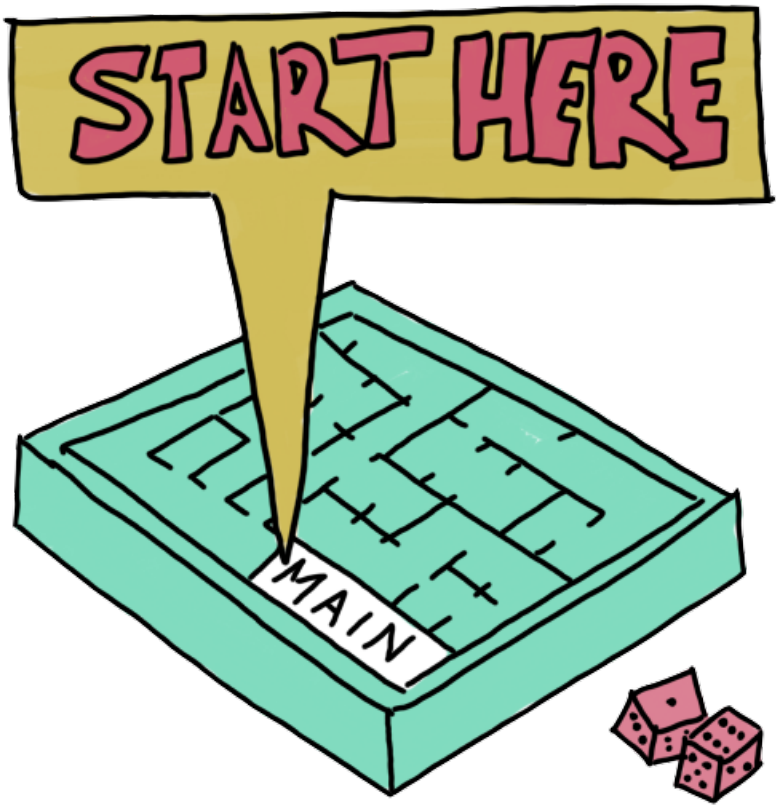
Did you notice you can use type signatures to let you work out what to plug in to what? They can be incredibly useful when programming.

## 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.

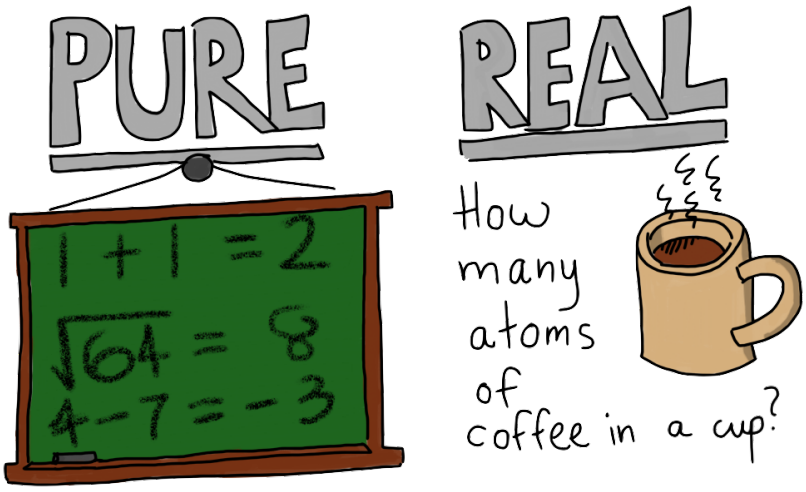
## 4.1 How a Haskell Program is Made

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
```

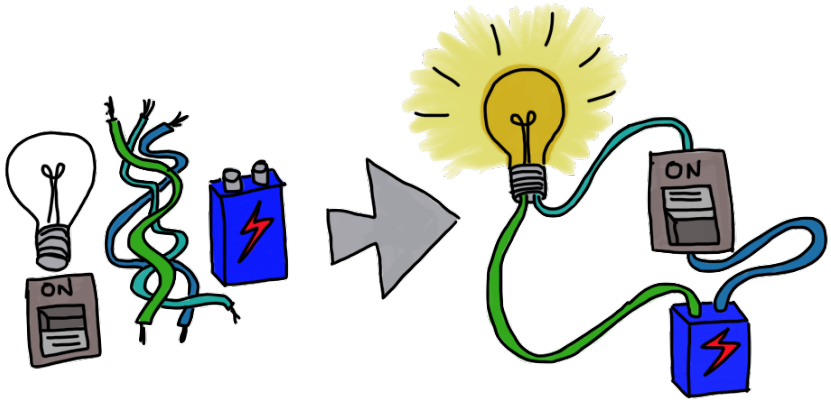
As we can see, the type of the function that can get the POSIX time (a type of time value) is an IO type, which shows that it is an action in the impure world. That means it will return a POSIXTime when it is **executed**.

So, `IO` actions can be **executed**, which is what happens when their `IO` be-

haviour 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

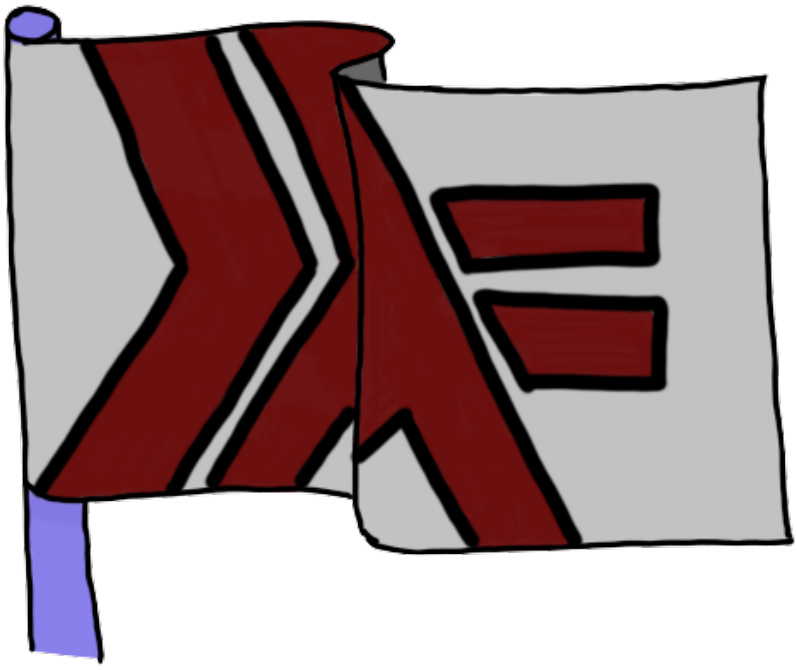
Let's use an analogy here: think about a simple electric circuit: a battery, some wires, a switch and a light globe. By themselves, all of these things are simple, "pure" things. The light and switch are slightly different than the others, but they are still just themselves. However, the switch can "do 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 possible types has only two values. They represent truth and falsehood. In Haskell, the values are `True` and `False`, and the type is `Bool`:

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

In later chapters, we'll see how important these two values are. We'll use them for checking what's true, and doing all sorts of other useful things. For now, because they're so simple — there's just two of them after all — we're going to use them to introduce how **functions** work in Haskell. We'll use a story to explain them, but all the Haskell in the story is real, and not actually magic.

## 5.1 A Story of Magic Coins

Let's imagine you own a bookshop, space travel is common, and you're on holidays on a foreign planet. You're looking for a good spot to read a book you brought with you called "The magic language of Haskell". The inhabitants of the planet you're visiting, Planet Boolean, have a fondness for milkshakes so you end up at a milk bar. The local currency are `Bool` coins and they're described in Haskell, as above.

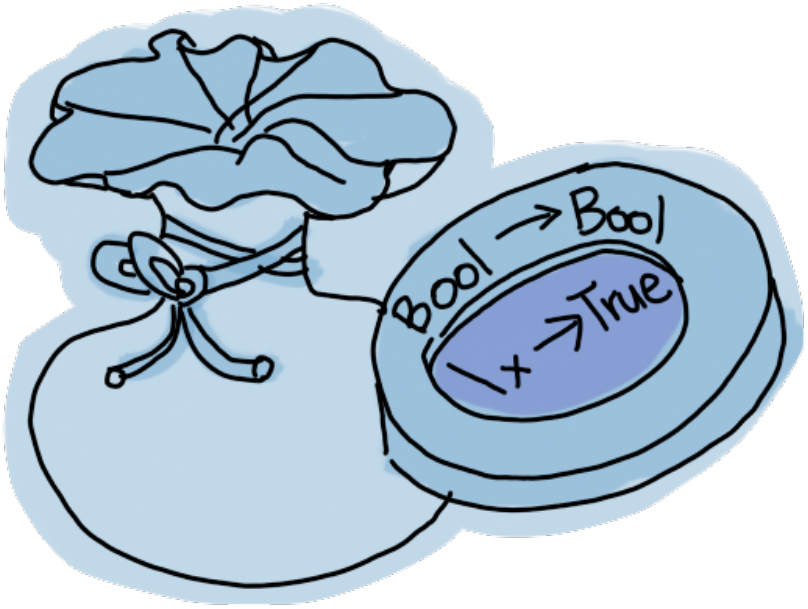
You just paid three `False` coins for a yangmei milkshake, and now you really want to play your favourite song on the jukebox as you read. You look at your money container as you wander towards the jukebox:


Drats! Mostly `False` coins. it seems the jukebox takes two `True` coins to play a song, and you only have one.

You spy a misshapen robot character at one end of the bar, muttering the word “lambda” over and over. Ordinarily you’d think it nothing more than a stressed-out robot, but the last chapter you read of your book described Haskell mage robots, and you recognise its robe. Maybe this robot can help you.

It puts down its milkshake as you explain your bother. After you finish, it pauses a moment then suddenly produces something from its cloak with a smile. It pushes the small, glowing bag in your direction, seeming very happy to be able to help.

"You pay for these by learning how they work" it says simply, refusing your offer of `False` coins as payment. Curious, you take the bag and look inside as you walk to the jukebox. You pick out one of the glowing coins from inside, and turn it over; it appears to be made of some quite mystical material you haven't seen before:




You notice an inscription on the coin. You recognise its arcane magical writing; it's written in Haskell, the ancient language of the robot mage folk:

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

You're a little familiar with this language yourself. Happily, you remember you just happen to have that beginner's guide to the Haskell language. Yes,

that was why you came in here in the first place, to read it!

You recall that the arrow symbol `->` on types usually indicates a function: that is, a type mapping from one **type** to another **type**.

You already know that `Bool` is the type of coins, which can be either `True` or `False`. You also already know that `::` indicates that the value on its left is of the type to its right, as in `True :: Bool` to say that the value `True` has the type `Bool`.

You consult the book again and realise `(\x -> True)` must be a value, and `Bool -> Bool` must be its type. That means it must be a function from `Bool` values to `Bool` values! You also realise the left and right arrows `->` mean a different thing to each other, because they're on different sides of `::`.

After a few minutes of rifling through the pages, you discover that `\x -> True` is an example of a **function literal**, or another term for this is a **lambda**. A lambda, you read, is an inline function definition. The parentheses are only needed here to make sure `:: Bool -> Bool` is referring to the whole lambda rather than just the `True` on the right side of it.

So, this coin **is** a function! This is one way to define a function in Haskell (we'll see many more in later chapters).

You realise that a function is a way of getting a completely new value from an existing value, sometimes using the existing value, sometimes not. It also says functions are values, too, so that means we can pass them into other functions, or return them from functions.

The syntax of a lambda, or the way it's written, has a backslash (`\`) followed by a variable name, (in this case `x`). The variable name is standing in for what-

ever we apply this function to, when it gets applied. This variable can be used in the area to the right of `->`.

However, it seems our lambda coin **isn't** using the `x` variable name at all on the right hand side of `->` in the lambda. If it was written as `\x -> x` then it would just give you back whatever value you put in, but `\x -> True` will always produce a fresh `True` value whenever you apply it to any `Bool` value; it ignores its input argument, the `x` variable, entirely. Functions are free to do this.

This is great, because it will do what you need. You can take your `False` coins and make new `True` ones from them! Those wily robot wizards are so clever, making such magic!

You take one of your coins, and you place it to the right of the magic coin, which will **apply the function to the value**, so the "magic" of function application will bind them together and produce a new value. POP!

The magic coin pops, they both disappear, and a brand new `True` coin appears! Slowly, as you look at it, you realise you just did something very silly, though. This is what is written on it:

```
True
-- from ((\x -> True) True)
```

The `Bool` coin you applied to the `\x -> True` lambda coin was that single `True` coin you already had! So it did nothing, effectively, and you wasted a lambda coin. You write this down in the margin of your book, for the future:

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

The `--` is called a code comment, and it's not a meaningful part of the code for doing anything, it's for writing notes or documentation 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 put parentheses around the lambda expression before you applied it to the coin. This is good, because if you didn't, Haskell would think it meant a function that takes an argument and then tries to use the first `True` as a function and apply it to the second `True` like this: `\x -> True True`, and that would cause Haskell to not compile your program, because it's meaningless; `True` isn't a function.

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.

Time to try with a `False` coin:

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

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 created coins `((\x -> True) True)` into another one of those lambda 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.

```
-- first we get a lambda coin:
-- (\x -> True)
-- then we get a fresh True coin we created from a lambda and a F
-- ((\x -> True) False)
-- now we apply the lambda coin, by putting it on the left of the
(\x -> True) ((\x -> True) False)
```

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 really complicated. The part on the right is the coin that resulted from shoving a `False` into the first lambda coin. Then we put that whole bracketed thing on the right of another lambda coin. It's correct, it just has lots of things going on!

Because `((\x -> True) False)` actually gets turned into `True` when it's evaluated, and can never mean anything else, we **could** have just substituted `True` instead, but that wouldn't explain the whole process.

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. The book has a definition for a magic coin:

```
lambdaCoin :: Bool -> Bool
lambdaCoin = \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 that argument won't be used. Fascinating, you think, as you happily sip your shake, humming along to the tune.

## 5.2 No Magic Necessary

We leave our story here, but we see that we can use the `lambdaCoin` function in the same way that we used `putStrLn` earlier, except because `lambdaCoin` has type `Bool -> Bool`, that is, because both its argument and output types are `Bool`, we can apply it as many times as we like, for example:

```
-- the function that always returns True no matter what Bool it g
lambdaCoin :: Bool -> Bool
lambdaCoin = \_ -> True

-- the value True, by applying the above lambdaCoin
-- function to the value False
newCoin :: Bool
newCoin = lambdaCoin False

-- the value True, by applying the above lambdaCoin
-- function to newCoin which is itself arrived at by
-- applying lambdaCoin to False
newCoinAgain :: Bool
newCoinAgain = lambdaCoin newCoin
```

```haskell
-- this is another way to write newCoinAgain,
-- but explicitly spelling out
-- all of the applications of lambdaCoin
newCoinAgain' :: Bool
newCoinAgain' = lambdaCoin (lambdaCoin False)
```

We can keep applying it as many times as we like. However, this particular function, `lambdaCoin`, is not extremely useful. It's only useful if you want to take any `Bool` and get back a `True`.

There's another way to write this function in Haskell. Let's take a look at it:

```haskell
lambdaCoin' :: Bool -> Bool
lambdaCoin' _ = 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 say any value at all, but the type signature requires it to be a `Bool` so it can actually only be one of `True` or `False`. Don't get too worried or confused about this, we will go into more gradual detail on how to write functions in upcoming chapters.

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

```haskell
lambdaCoin'' :: Bool -> Bool
lambdaCoin'' True  = True
lambdaCoin'' False = True
```

There is no reason to do this with our trivial example, but this shows you another way to write functions: we're matching on each input value.

Another thing we could do is write a function that flips a boolean value to the other value. This is actually extremely useful, and built into Haskell already as the function named `not`, but we'll make our own, to show you how easy it is to write and 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`. Instead, 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 example. This one will actually return the `lambdaCoin` function! We know functions are values, just like `Bool` values or `String` values or any other type of values. That means we can take them as arguments, and give them back as results of functions.

What is 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 its type is `Bool -> (Bool -> Bool)`

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

So we take any `Bool` and return `lambdaCoin` which is itself the function `\_ -> True`. This is a function that takes two arguments!

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

```
lambdaCoinTakesTwo :: Bool -> (Bool -> Bool)
lambdaCoinTakesTwo = \_ -> lambdaCoin
```

The type actually doesn't need parentheses on it, because they naturally group up that way. Let's remove parentheses from the signature, and spell `lambdaCoin` out as a lambda.

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

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

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

-- using just one lambda
lambdaCoinTakesTwo'2 :: Bool -> Bool -> Bool
lambdaCoinTakesTwo'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 `lambdaCoin` 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`.

```
lambdaCoinTakesTwo False False -- will be True
lambdaCoinTakesTwo True False -- will be True
lambdaCoinTakesTwo False True -- will be True
lambdaCoinTakesTwo True True -- will be True
```

This is a pretty pointless function. However, now we've seen how to use functions of two arguments, let's switch the definition into normal function syntax, and then we'll look at a very similar and actually very useful function:

```
-- a more normal way of defining the function
lambdaCoinTakesTwo'' :: Bool -> Bool -> Bool
lambdaCoinTakesTwo'' _ _ = True
```

What have we done here? This is another way to define the same function, only instead of writing it as a lambda, we've written it using **function definition syntax**. Don't worry too much if this is confusing, we'll introduce this more slowly and properly in the next chapters. For now, just know that it's another way of defining the same function as `lambdaCoinTakesTwo`.

## 5.4 A Dip into Logic

Ok, so what if we wanted a function of two boolean values that tells us if any one of the two values is **True**? For example, at a carnival, maybe the ride attendant wants to say "you can ride the scary ride, but only if you're the right height, or you're an adult". Let's write this logic in some Haskell:

```
tallEnough = False
isAdult = False
canRideScaryRide = False
```

Let's say instead of just writing down the value for `canRideScaryRide` directly, we want to have Haskell figure it out instead; and as we know, they can only ride it if they're either tall enough, or they're an adult, or both, so we want to base the value on `tallEnough` and `isAdult`.

Here's how we could define a function to help us do this:

```
isEitherTrue :: Bool -> Bool -> Bool
isEitherTrue False False = False
isEitherTrue _     _     = True
```

So this is another function definition. Here we're making a new function called `isEitherTrue`. We're **pattern matching** on the arguments. In the first line, if both the arguments match as `False`, the result will equal `False`, and in the next line it says for every other combination of argument values (using the underscores to match anything), the result is going to be `True`. (That is: when either one is `True`, or both are `True`). Again, if this is confusing, just make a note to come back and take a look after reading the next chapters.

Let's see it in action:

```
tallEnough = False
isAdult = False
canRideScaryRide = isEitherTrue tallEnough isAdult -- this is Fal
```

Neat, isn't it? This `isEitherTrue` we've written is a very useful function, and programmers use it all the time, though the built in one is named differently. We'll see that later.

Now we can change either of the dependant values, and know that `canRideScaryRide` will get the right value without us having to do anything extra to its definition.

```
tallEnough = True
isAdult = False
canRideScaryRide = isEitherTrue tallEnough isAdult -- This will b
```

Because this is such a useful function, it's actually already in Haskell as the **logical or operator**, and it works like this:

```
tallEnough = False
isAdult = False
canRideScaryRide = tallEnough || isAdult
```

There are other logical operators, too, and we'll see them later.

## 5.5 A Hint of Curry

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 using more functions, we can get more arguments. Of course, as we've seen, Haskell has a lot of nice syntax to make it easier to write functions of more than one argument because it's such a common need.

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.

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

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

increment :: Int -> Int
increment = plus 1

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.

## 5.6 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:

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

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

```
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:

```
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.

Next, see if you can write a function similar to `isEitherTrue` but that will only return `True` when **both** are `True`. Call this function `areBothTrue` and try to use it to write a value for `canRideVeryScaryRide` which is only for tall adults to ride. Try it out with different values for the two boolean values.

If you've already learned another programming language, then this particular point to do with currying will likely be fairly difficult for you to understand. 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 has less parts to it. Of course, if you've already learned a complex language 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 as you now know and understand, 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 End of the Free Sample

## 6.1 Thanks!

We really hope you've enjoyed this free sample of the book.

Please consider purchasing the full book, and letting as many people know about our works as possible.

This will enable us to make more great works like this for you in the future!

— GetContented

`http://happylearnhaskelltutorial.com`