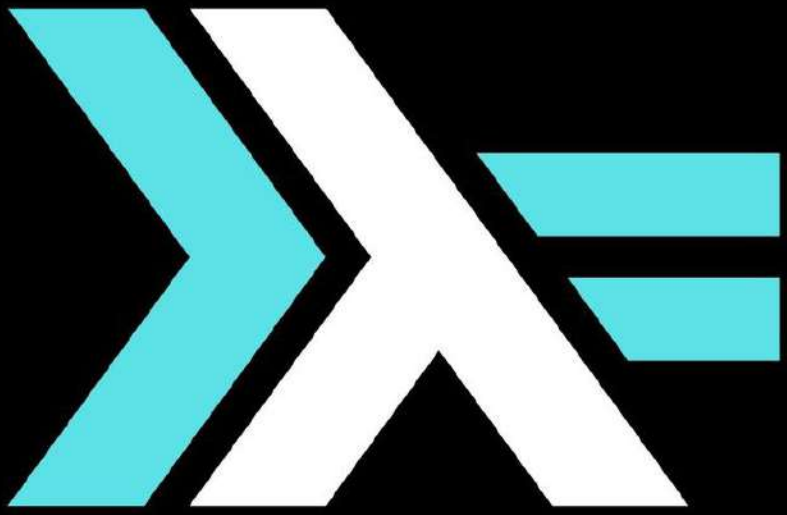
MEM LNC

Haskell

The Ultimate Beginner's Guide to Learn Haskell Programming Step by Step



BY
CLAUDIA ALVES

Haskell

The Ultimate Beginner's Guide to Learn Haskell Programming Step by Step

1st edition

2020

By Claudia Alves

"Programming isn't about what you know; it's about what you can figure out." ***- Chris Pine***

memlnc

Introduction

A balance of flexible and inflexible qualities make Haskell a fascinating programming language to learn and use.

First, the Haskell programming language is not named after Eddie Haskell, the sneaky double-dealing neighbor kid in the ancient TV sitcom, Leave It To Beaver.

Haskell is named after Haskell Brooks Curry, an American mathematician and logician. If you don't know, logicians create models to describe and define human reasoning, for example, problems in mathematics, computer science, and philosophy. Haskell's main work was in combinatory logic, a notation designed to eliminate the need for variables in mathematical logic. Combinatory logic captures many key features of computation and, as a result, is useful in computer science. Haskell has three programming languages named after him: Haskell, Brooks, and Curry.

Haskell the language is built around functions, useful blocks of code that do specific tasks. They are called and used only when needed.

Another interesting feature of functional languages like Haskell: functions are treated as values like integers (numbers) and strings. You can add a function to another function the way you can add an integer to an integer, 1 + 1 or 35 + 53. Perhaps the best way to describe this quality is a spreadsheet: in a cell in the spreadsheet, you can add numbers as well as a combination of functions to work on numbers. For example, you might specify each number in cells 1-10 be added up as a sum. In Excel, at least, you also can use SUMIF to look for a pattern in cells 1-10 and, if the pattern is found, perform an action on any cells with the pattern.

What Makes Haskell Special?

Technically, Haskell is a general-purpose functional programming language with non-strict semantics and strong static typing. The primary control construct is the function. (Say that fast ten times!) Here's what it means:

- Every language has a strategy to evaluate when to process the input arguments used in a call to a function. The simplest strategy is to evaluate the input arguments passed then run the function with the arguments. Non-strict semantics means the input arguments are not evaluated unless the arguments passed into the function are used to evaluate what is in the body of the function.

- Programming languages have rules to assign properties — called a type — to the components of the language: variables, functions, expressions, and modules. A type is a general description of possible values the variable, function, expression, or module can store. Typing helps minimize bugs, for example, when a calculation uses a string ("house" or "cat") instead of a number (2 or 3). Strong static typing evaluates the code before runtime, when the code is static and possibly as code is written.

- The order in which statements, instructions and functions are evaluated and executed determines the results of any piece of code. Control constructs define the order of evaluation. Constructs use an initial keyword to flag the type of control structure used. Initial keywords might be "if" or "do" or "loop" while final keywords might be "end if" or "enddo" or "end loop". Instead of a final keyword, Haskell uses indentation level (tabs) or curly brackets, or a mix, to indicate the end of a control structure.

Perhaps what makes Haskell special is how coders have to think when they use the language. Functional programming languages work in very different ways than imperative languages where the coder manages many low-level details of what happens in their code and when. While it is true all languages have things in common, it's also true languages are mostly functional or

mostly imperative, the way people are mostly right handed or left handed. Except functional programming languages require a different way of thinking about software as you code.

Other features that make Haskell interesting:

- **Strong data typing (evaluating properties of all inputs into a function) is combined with polymorphism; a function to sort numbers also can be used to sort strings of text. In some languages, you would have to code two or more functions, one for each data type.**

- **Lazy evaluation (one of my favorite coding terms!) allows the result of one function/task to be handed to another function/task on the same line of code. For example, the command can search a file for all instances of a string then pass the results to be printed to the computer screen. Functions that can take other functions as arguments or return them as results also are called higher order functions.**

- **No side effects. In other languages, code can affect the state of the computer and application, for example, writing to a file. Haskell strictly limits these side effects which, in turn, makes Haskell applications less prone to errors.**

- **Haskell uses monads, a structure that works like an assembly line where every stop on the line performs a different task. This allows Haskell to separate side effects as a distinct activity apart from any function, for example, logging any errors as a function performs tasks on its data inputs.**

Building from small bits of code, each bit tightly contained and testable.

How is Haskell Used?

As a functional programming language, Haskell has benefits like shorter development time, cleaner code, and high reliability. The tight control of side effects also eliminates many unforeseen interactions within a code base. These features are especially of interest to companies who must build software with high fault tolerances, for example, defense industries, finance, telecommunications, and aerospace.

However, Haskell also is used in web startups where functional programming might work better than imperative programming. Apparently Facebook, Google, NVIDIA, and other companies use Haskell to build internal tools used in their software development and IT environments. Even a lawn mower manufacturer in Kansas uses Haskell to build and distribute their mowers. And the New York Times recently used Haskell to build an image processing tool for the 2013 New York Fashion week.

So what is Haskell?

Haskell is a **purely functional programming language** . In imperative languages we obtain results by giving the computer a sequence of tasks that it will then execute. While running them, you can change state. For example, we set the variable to 5, perform some tasks, and then change the value of the previous variable. These languages have flow control structures to carry out certain actions several times (for , while ...). With purely functional programming we do not tell the computer what it has to do, but rather, we say how things are. The factorial of a number is the product of all the numbers from 1 to that number, the sum of a list of numbers is the first number plus the sum of the rest of the list, etc. We express the form of the functions. Also we can't set a variable to something and then set it to something else. If we say that a is 5, then we cannot say that it is something else because we have just said that it is 5. Are we liars? Thus, in purely functional languages, a function has no side effects. The only thing a function can do is calculate and

return something as a result. At first this may seem like a limitation but in reality it has some good consequences: if a function is called twice with the same parameters, we will always get the same result. We call this *referential transparency* and it not only allows the compiler to reason about the behavior of a program, but it also allows us to easily deduce (and even demonstrate) that a function is correct and thus be able to build more complex functions by joining simple functions.

Haskell is **lazy** . That is, unless we tell you otherwise, Haskell will not execute functions or calculate results until you are really forced to. This works very well in conjunction with referential transparency and allows us to view programs as a series of data transformations. It even allows us to do cool things like infinite data structures. Let's say we have a list of immutable numbers `xs = [1,2,3,4,5,6,7,8]` and a `doubleMe` function that multiplies each item by 2 and returns a new list. If we wanted to multiply our list by 8 in an imperative language if we did `doubleMe (doubleMe (doubleMe (xs)))` , the computer would probably `loop` through the list, make a copy and return the value. Then it would go through the list two more times and return the final value. In lazy language, calling `doubleMe` with an `unforced` list to display the value ends up with a program telling you "Sure, sure, then I'll do it!". But when you want to see the result, the first `doubleMe` tells the second one that he wants the result, now! The second says the same to the third and the latter reluctantly returns a duplicate 1, which is a 2. The second receives it and returns a 4 to the first. The first one sees the result and says that the first item in the list is an 8. In this way, the computer only makes a journey through the list and only when we need it. When we want to calculate something from initial data in lazy language, we just have to take this data and transform and mold it until it resembles the result we want.

Haskell is a statically **typed** language . When we compile a program, the compiler knows which pieces of the code are integers, which are text strings, etc. Thanks to this a lot of possible errors are caught at compile time. If we try to add a number and a text string, the compiler will scold us. Haskell uses a fantastic type system that has type inference. This means that we don't have to explicitly tag each piece of code with a type because the type system can

intelligently deduce it . Type inference also allows our code to be more general, if we have created a function that takes two numbers and adds them together and we do not explicitly set their types, the function will accept any pair of parameters that act as numbers.

Haskell is elegant and concise. It is because it uses high-level concepts. Haskell programs are typically shorter than the imperative equivalents. And short programs are easier to maintain than long ones, and they have fewer errors.

Haskell was created by some very smart guys (all of them with their respective doctorates). The project to create Haskell began in 1987 when a committee of researchers agreed to design a revolutionary language. In 2003 the Haskell report was published, thus defining a stable version of the language.

What do you need to start

A Haskell text editor and compiler. You probably already have your favorite text editor installed so we're not going to waste your time on this. Right now, Haskell's two main compilers are GHC (Glasgow Haskell Compiler) and Hugs. For the purposes of this guide we will use GHC. I will not cover many details of the installation. In Windows it is a matter of downloading the installer, pressing "next" a few times and then restarting the computer. In Debian-based Linux distributions it can be installed with `apt-get` or by installing a `deb` package . In MacOS it is a matter of installing a `dmg` or using `macports` . Whatever your platform, here is more information.

GHC takes a script from Haskell (they usually have the *.hs* extension) and compiles it, but it also has an interactive mode which allows us to interact with these scripts. We can call the functions of the scripts that we have loaded and the results will be shown immediately. It is much easier and faster to learn instead of having to compile and run the programs over and over again. Interactive mode is executed by typing `ghci` from your terminal. If we have defined some functions in a file called, say, `myFunctions.hs` , we can load those functions by typing `:l myFunctions` , as long as `myFunctions.hs` is in the same directory where `ghci` was invoked . If we modify the *.hs* script and want to observe the changes we have to rerun `:l myFunctions` or run `:r` which is

equivalent since it reloads the current script. We will work defining some functions in a *.hs* file , we *load them* and spend time playing with them, then we will modify the *.hs* file by *reloading* it and so on. We will follow this process throughout the guide.

Starting
Ready, Set, Go!

Alright, let's get started! If you are that kind of bad person who doesn't read the introductions and you have skipped it, you may want to read the last section of the introduction because it explains what you need to follow this guide and how we are going to work. The first thing we are going to do is run GHC in interactive mode and use some functions to get used to it a little. Open a terminal and type ghci . You will be greeted with a greeting like this:

```
GHCi, version 7.2.1: http://www.haskell.org/ghc/:? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Congratulations, you came from GHCi! Here the pointer (or *prompt*) is Prelude> but since it gets longer as we load modules during a session, we are going to use ghci> . If you want to have the same pointer execute : set prompt "ghci> " .

Here we have some simple arithmetic.

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

It is self explanatory. We can also use several operations on the same line so that all the rules of precedence that we all know are followed. We can use parentheses to use explicit precedence.

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

Very interesting, huh? Yes, I know not, but be patient. A small difficulty to keep in mind occurs when we deny numbers, it will always be better to surround negative numbers with parentheses. Doing something like 5 * -3 will make GHCi angry, however 5 * (-3) will work.

Boolean algebra is also quite simple. As you probably know, && represents the logical **AND** while || represents the logical **OR** . not denies True to False and vice versa.

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

The equality check is done like this:

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

What if we do something like 5 + "text" or 5 == True ? Well, if we try the first one we get this friendly error message:

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

GHCi is telling us that "text" is not a number and therefore does not know how to add it to 5. Even if instead of "text" it were "four" , "four" , or "4" ,

Haskell would not consider it as a number. + expects its left and right sides to be numbers. If we try to perform True == 5 , GHCi would tell us that the types do not match. While + works only with things that are considered numbers, == works with anything that can be compared. The trick is that both must be comparable to each other. We can't compare speed with bacon. We'll take a more detailed look at the types later. Note: we can do 5 + 4.0 because 5 does not have a specific type and can act as an integer or as a floating point number. 4.0 cannot act as an integer, so 5 is the only one that can be adapted.

You may not know it, but we have been using functions all this time. For example, * is a function that takes two numbers and multiplies them. As you have already seen, we call him making a sandwich on him. We call this infix functions. Many functions that are not used with numbers are prefixes. Let's see some of them.

Functions are normally prefixes so from now on we are not going to say that a function is in prefix form, we will just assume it. In many imperative languages functions are called by writing their names and then writing their parameters in parentheses, usually separated by commas. In Haskell, functions are called by typing their name, a space, and their parameters, separated by spaces. For starters, let's try calling one of Haskell's most boring functions.

```
ghci> succ 8
9
```

The succ function takes anything that has a successor defined and returns that successor. As you can see, we have simply separated the function name and its parameter by a space. Calling a function with multiple parameters is just as easy. The min and max functions take two things that can be put in order (like numbers!) And return one of them.

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

The application of functions (calling a function by putting a space after it and then writing its parameters) has the highest priority. Said with an example,

these two sentences are equivalent:

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

However, if we had wanted to obtain the successor of the product of the numbers 9 and 10, we could not have written succ 9 * 10 because we would have obtained the successor of 9, which would have been multiplied by 10, obtaining 100. We have to write succ (9 * 10) to get 91.

If a function takes two parameters we can also call it as an infix function by surrounding it with open accents. For example, the div function takes two integers and performs an integer division between them. Doing div 92 10 would get 9. But when we call it that, there may be some confusion as to what number is doing the division and which is being divided. So we call it as an infix function making 92 `div` 10 , thus making it clearer.

People who already know some imperative language tend to cling to the idea that parentheses indicate an application of functions. For example, in C, you use parentheses to call functions like foo () , bar (1) , or baz (3, "haha") . As we have said, spaces are used to apply functions in Haskell. So these functions in Haskell would be foo , bar 1 and baz 3 "haha" . If you see something like bar (bar 3) it does not mean that bar is called with bar and 3 as parameters. It means that we first call the function bar with 3 as a parameter to get a number, and then call bar again with that number. In C, this would be something like bar (bar (3)) .

The first small functions

In the previous section we got a basic idea of how to call the functions. Now let's try to make ours! Open your favorite text editor and paste this function that takes a number and multiplies it by two.

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

Functions are defined similarly to what they are called. The function name is followed by the parameters separated by spaces. But, when we are defining functions, there is an = and then we define what the function does. Save this as baby.hs or as you like. Now navigate to where you saved it and run ghci

from there. Once inside GHCi, write : l baby . Now that our code is loaded, we can play with the function that we have defined.

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

Since + works with integers just as well as with floating-point numbers (actually anything that can be considered a number), our function also works with any number. We are going to make a function that takes two numbers, multiplies each of them by two and then adds both.

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

Simple. We could also have defined it as doubleUs x y = x + x + y + y . Both forms produce very predictable results (remember to add this function in the baby.hs file , save it and then execute : l baby inside GHCi).

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

As you can deduce, you can call your own functions within the functions you do. With this in mind, we could redefine doubleUs as:

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

This is a simple example of a normal pattern that you will see throughout Haskell. Create small functions that are obviously correct and then combine them into more complex functions. In this way you will also avoid repeating yourself. What if some mathematicians discover that 2 is actually 3 and you have to change your program? You can simply redefine doubleMe to be x + x + x and how doubleUs calls doubleMe will automatically work in this strange world where 2 is 3.

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

Now we are going to create a function that multiplies a number by 2 but only if that number is less than or equal to 100, because the numbers greater than

100 are already large enough on their own.

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

We have just introduced the Haskell if statement . You are probably already familiar with the if statement from other languages. The difference between Haskell's if statement and that of imperative languages is that the else part is mandatory. In imperative languages we can skip a few steps if a condition is not satisfied, but in Haskell each expression or function must return a value. We could also have defined the if statement on a single line but it seems a bit more readable that way. Another thing about the if statement in Haskell is that it is an expression. Basically an expression is a piece of code that returns a value. 5 is an expression because it returns 5, 4 + 8 is an expression, x + y is an expression because it returns the sum of x and y . Since the else part is mandatory, an if statement will always return something and is therefore an expression. If we want to add one to each number that is produced by the previous function, we can write its body like this.

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

If we had omitted the parentheses, I would have only added one if x was not greater than 100. Look at the ' at the end of the function name. That apostrophe has no special meaning in Haskell's syntax. It is a valid character to be used in the name of a function. We usually use ' to denote the strict version of a function (one that is not lazy) or a small modified version of a function or variable. Since ' is a valid character for functions, we can do things like this.

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

There are two things that remain to be highlighted. The first is that the name of this function does not begin with capital letters. This is because functions cannot start with an uppercase letter. We will see why a little later. The second is that this function does not take any parameters, we usually call it a definition (or a name). Since we can't change definitions (and functions) after we've defined them, conanO'Brien and the string "It's a-me, Conan O'Brien!" they can be used interchangeably.

An introduction to the lists

Like real-life shopping lists, lists in Haskell are very helpful. It is the most widely used data structure and can be used in different ways to model and solve a lot of problems. The lists are VERY important. In this section we will take a look at the basics about lists, text strings (which are lists) and intensional lists.

In Haskell, lists are a **homogeneous** data structure . Stores multiple items of the same type. This means that we can create an integer list or a character list, but we cannot create a list that has a few integers and a few other characters. And now, a list!

Note

We can use the let keyword to define a name in GHCi. Doing let a = 1 inside GHCi is equivalent to writing a = 1 to a file and then loading it.

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

As you can see, the lists are defined by square brackets and their values are separated by commas. If we tried to create a list like this [1,2, 'a', 3, 'b', 'c', 4] , Haskell would warn us that the characters (which by the way are declared as a character in single quotes) are not numbers. Speaking of characters, strings are simply lists of characters. "hello" is just a syntactic alternative to ['h', 'e', 'l', 'l', 'o'] . Since the strings are lists, we can use the functions that operate with lists on them, which is really useful.

A common task is to concatenate two lists. Which we did with the ++ operator .

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

Be careful when using the ++ operator repeatedly on long strings. When we

concatenate two lists (even if we add a list of one item to another list, for example [1,2,3] ++ [4] , internally Haskell has to loop through the entire list from the left side of the ++ operator . This is not a problem when working with lists that are not too big. But concatenating something at the end of a list that has fifty million elements will take a while. However, concatenating something at the beginning of a list using the operator : (also called operator cons) is instantaneous.

```
ghci> 'U': "n black cat"
"A black cat"
ghci> 5: [1,2,3,4,5]
[5,1,2,3,4,5]
```

Notice that : it takes a number and a list of numbers or a character and a list of characters, while ++ takes two lists. Even if you add an item to the end of the lists with ++ , you have to surround it with square brackets so that it becomes a single item list.

```
ghci> [1,2] ++ 3
<interactive>: 1: 10:
    No instance for (Num [a0])
      arising from the literal `3 '
      [...]

ghci> [1,2] ++ [3]
[1,2,3]
```

[1,2,3] is a 1: 2: 3 syntactic alternative : [] . [] is an empty list. 3 If we put it with : we obtain [3] , and if we put 2 to get this [2,3] .

Note

[] , [[]] and [[], [], []] are different things from each other. The first is an empty list, the second is a list containing one item (an empty list), and the third is a list containing three items (three empty lists).

If we want to get an item from the list knowing its index, we use !! . Indexes start with 0.

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

But if we try to get the sixth item in a list that only has four items, we will get

an error, so be careful.

Lists can also contain lists. These can also contain lists that contain lists, that contain lists ...

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

The lists within the lists can have different sizes but cannot have different types. In the same way that you cannot contain characters and numbers in a list, you cannot contain lists that contain character lists and number lists either.

The lists can be compared if the elements they contain can be compared. When we use < , <= , > , and > = to compare lists, they are compared in lexicographic order. The heads are first compared. Then the second elements are compared and so on.

What else can we do with the lists? Here are some basic functions that can operate with lists.

- head takes a list and returns its head. The head of a list is basically the first element.
- ghci> head [5,4,3,2,1]
- 5
- tail takes a list and returns its tail. In other words, cut off the head of the list.
- ghci> tail [5,4,3,2,1]
- [4,3,2,1]
- last takes a list and returns its last element.
- ghci> last [5,4,3,2,1]
- one
- init takes a list and returns the entire list except its last element.
- ghci> init [5,4,3,2,1]
- [5,4,3,2]

If we imagine the lists as monsters, they would be something like:

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

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

Oh we broke it! If there is no monster, there is no head. When we use head , tail , last and init we must be careful not to use empty lists with them. This error cannot be caught at compile time so it is always a good practice to take precautions before telling Haskell to return you some items from an empty list.

- length takes a list and obviously returns its size.
- ghci> length [5,4,3,2,1]
- 5
- null checks if a list is empty. If it is, it returns True , otherwise it returns False . Use this function instead of xs == [] (if you have a list called xs).
- ghci> null [1,2,3]
- False
- ghci> null []
- True
- reverse reverses a list.
- ghci> reverse [5,4,3,2,1]
- [1,2,3,4,5]
- take takes a number and a list and extracts said number of elements from a list. Observe.
- ghci> take 3 [5,4,3,2,1]
- [5,4,3]
- ghci> take 1 [3,9,3]
- [3]
- ghci> take 5 [1,2]
- [1,2]
- ghci> take 0 [6,6,6]
- []

Note that if we try to take more elements than there are in a list, it simply returns the list. If we take 0 elements, we get an empty list.

- drop works similarly, except that it removes a number of items from the beginning of the list.
- ghci> drop 3 [8,4,2,1,5,6]
- [1,5,6]
- ghci> drop 0 [1,2,3,4]
- [1,2,3,4]
- ghci> drop 100 [1,2,3,4]
- []
- maximum takes a list of things that can be put in some sort of order and returns the largest element.
- minimum returns the smallest.
- ghci> minimum [8,4,2,1,5,6]

- one
- ghci> maximum [1,9,2,3,4]
- 9

- sum takes a list of numbers and returns their sum.
- product takes a list of numbers and returns your product.

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

- elem takes a thing and a list of things and tells us if that thing is an element of the list. This function is normally called infixed because it is easier to read.

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

These were a few basic functions that operate with lists. We will see more functions that operate with lists later.

Texas ranges

What if we want a list with all the numbers between 1 and 20? Yes, we could just write them all but obviously this is not a solution for those who are looking for good programming languages. Instead, we will use ranges. Ranges are a way to create lists that contain an arithmetic sequence of enumerable elements. The numbers can be numbered. One, two, three, four, etc. Characters can also be numbered. The alphabet is an enumeration of characters from A to Z. The names are not enumerable. What comes after "Juan"? No idea.

To create a list containing all the natural numbers from 1 to 20 we simply write [1..20] . It is equivalent to writing [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] and there is no difference between typing one or the other except that manually typing a long sequence of enumerables is pretty stupid.

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

"ABCDEFGHIJKLMNOPQRSTU VWXYZ"
ghci> ['K' .. 'Z']
"KLMNOPQRSTUVWXYZ"

We can also specify the number of steps between elements of a range. What if we want all the even numbers from 1 to 20? Or every third number?

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

It is a matter of separating the first two elements with a comma and then specifying the upper limit. Although they are smart, the step ranges are not as smart as some people expect them to be. You cannot type [1,2,4,8,16..100] and expect to get all powers of 2. First because only one step can be specified. And second, because the sequences that are not arithmetic are ambiguous if we only give a few initial elements.

To get a list with all the numbers from 20 to 1 we cannot use [20..1] , we must use [20,19..1] .

Be careful when using floating point numbers with ranges! These are not entirely accurate (by definition), and their use with ranges may give some unexpected results.

ghci> [0.1, 0.3 .. 1]
[0.1.0.3.0.5.0.7.0.8999999999999999.1.0999999999999999]

My advice is not to use ranges with floating point numbers.

We can also use ranges to create infinite lists simply by not indicating an upper limit. Later we will focus more on the infinite lists. For now, let's examine how we would get the first 24 multiples of 13. Yes, we can use [13,26..24 * 13] . But there is a better way: take 13 [13,26 ..] . Since Haskell is lazy, he won't try to evaluate the infinite list immediately because it would never end. It will wait to see what you want to get from the infinite list. In this case he sees that we only want the first 24 elements and he evaluates them with pleasure.

Now, a couple of functions that generate infinite lists:

- cycle takes a list and creates an infinite cycle of equal lists. If we tried to display the result it would never end so you have to cut it somewhere.
- ghci> take 10 (cycle [1,2,3])
- [1,2,3,1,2,3,1,2,3,1]

- ghci> take 12 (cycle "LOL")
- "LOL LOL LOL"

- repeat takes an element and produces an infinite list containing that single repeated element. It is like cycling a list with a single item.

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

Although here it would be simpler to use the replicate function , since we know the number of elements in advance. replicate 3 10 returns [10,10,10] .

I'm an intensional list

If you ever had math classes, you probably saw some intensively defined set, defined from other more general sets. An intensively defined set containing

the first ten even natural numbers would be . The part before the

separator is called the output function, it is the variable, it is the

input set and it is the predicate. This means that the set contains all the doubles of the natural numbers that fulfill the predicate.

If we wanted to write this in Haskell, we could use something like take 10 [2,4 ..] . But what if we didn't want the doubles of the first ten natural numbers, but something more complex? For this we can use intensional lists. Intensive lists are very similar to intensively defined sets. In this case, the intensional list we should use would be [x * 2 | x <- [1..10]] . x is extracted from [1..10] and for each element of [1..10] (which we have linked to x) we calculate its double. Its result is:

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

As we can see, we obtain the desired result. Now we are going to add a condition (or a predicate) to this intensional list. The predicates go after the part where we bind the variables, separated by a comma. Let's say we only want elements that have a double greater than or equal to twelve:

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

Well it works. What if we wanted all the numbers from 50 to 100 whose remainder when divided by 7 was 3? Easy:

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

A complete success! Removing items from the list using predicates is also known as **filtering** . We take a list of numbers and filter it using predicates. Another example, let's say we want an intensional list to replace each odd number greater than ten with "BANG!" and each odd number less than ten for "BOOM!". If a number is not odd, we leave it off the list. For convenience, we are going to put the intensional list inside a function to make it easily reusable.

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

The last part of understanding is the predicate. The odd function returns True if we pass it an odd number and False with an even number. The element is included in the list only if all predicates are evaluated to True .

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

We can include several predicates. If we wanted all the elements from 10 to 20 that were not 13, 15 or 19, we would do:

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

Not only can we have multiple predicates in an intensional list (an element must satisfy all predicates to be included in the list), but we can also extract elements from multiple lists. When we extract elements from several lists, all possible combinations of these lists are produced and joined according to the output function that we supply. An intensional list that extracts elements from two lists whose lengths are 4, will have a length of 16 elements, as long as we do not filter them. If we have two lists, [2,5,10] and [8,10,11] and we want the product of all possible combinations between them, we can use something like:

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

As expected, the length of the new list is 9 What if we wanted all possible products whose value is greater than 50?

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

How about an intensional list that combines a list of adjectives with a list of nouns? Just to rest easy ...

```
ghci> let nouns = ["frog", "zebra", "goat"]
ghci> let adjetives = ["lazy", "angry", "intriguing"]
ghci> [noum ++ "" ++ adjetive | noum <- nouns, adjective <- adjetives]
["lazy frog", "angry frog", "intriguing frog", "lazy zebra",
"angry zebra", "intriguing zebra", "lazy goat", "angry goat",
"intriguing goat"]
```

Already! We are going to write our own version of length . We will call it length ' .

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

_ means that we don't care what we are going to extract from the list, so instead of writing the name of a variable that we would never use, we simply write _ . The function replaces each item in the original list with 1 and then adds them together. This means that the resulting sum will be the size of our list.

A reminder: since strings are lists, we can use intensional lists to process and produce strings. For example, a function that takes strings and removes everything except capital letters from them would be something like this:

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

Quick tests:

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "noMEGUSTANLASRANAS"
"MEGUSTANLASRANAS"
```

In this case the predicate does all the work. It says that the element will be included in the list only if it is an element of [A..Z] . It is possible to create

nested intensional lists if we are working with lists that contain lists. For example, given a list of number lists, we will remove the odd numbers without flattening the list:

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

We can write the intensional lists on several lines. If we are not using GHCi it is better to divide the intensional lists into several lines, especially if they are nested.

Tuples

In some ways, tuples are similar to lists. Both are a way to store multiple values in a single value. However, there are a few fundamental differences. A list of numbers is a list of numbers. That is its type and it doesn't matter if it has a single item or an infinite number of them. Tuples, however, are used when you know exactly how many values have to be combined and their type depends on how many components they have and the type of these components. Tuples are denoted in parentheses and their values are separated by commas.

Another key difference is that they do not have to be homogeneous. Unlike lists, tuples can contain a combination of values of different types.

Think about how we would represent a two-dimensional vector in Haskell. One way would be using lists. It could work. So what if we wanted to put several vectors into a list representing the points of a two-dimensional figure? We could use something like [[1,2], [8,11], [4,5]] . The problem with this method is that we could also do things like [[1,2], [8,11,5], [4,5]] since Haskell has no problem with it, it's still a list of number lists but it doesn't make any sense. But a size 2 tuple (also called a pair) has its own type, which means you can't have multiple pairs and a triple (a size 3 tuple) in a list, so let's use these. Instead of using square brackets around the vectors we use parentheses: [(1,2), (8,11), (4,5)] . What if we try to create a shape like [(1,2), (8,11,5), (4,5)] ? Well, we would get this error:

```
Couldn't match expected type `(t, t1) '
```

against inferred type `(t2, t3, t4) '
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it ': it = [(1, 2), (8, 11, 5), (4, 5)]

It is telling us that we have tried to use a double and a triple in the same list, which is not allowed since the lists are homogeneous and a double has a different type than a triple (even though they contain the same type of values). We also can't do something like [(1,2), ("one", 2)] since the first item in the list is a tuple of numbers and the second is a tuple of a string and a number. Tuples can be used to represent a wide variety of data. For example, if we want to represent the name and age of someone in Haskell, we can use the triple: ("Christopher", "Walken", 55) . As we have seen in this example, tuples can also contain lists.

We use tuples when we know in advance how many components of some data we must have. Tuples are much more rigid than lists since for each size they have their own type, so we cannot write a general function that adds an element to a tuple: we have to write a function to add duplicates, another function to add triples, another function to add quadruples, etc.

While there are unit lists, there are no unit tuples. It really doesn't make much sense if you think about it. A unit tuple would simply be the value it contains and would not provide us with anything useful.

Like lists, tuples can be compared if their elements can be compared. Only we cannot compare two tuples of different sizes while we can compare two lists of different sizes. Two useful functions to operate with pairs are:

- fst takes a pair and returns its first component.
- ghci> fst (8.11)
- 8
- ghci> fst ("Wow", False)
- "Wow"
- snd takes a pair and returns its second component. Surprise!
- ghci> snd (8,11)
- eleven
- ghci> snd ("Wow", False)
- False

Note

These functions only operate on pairs. They will not work on triples, quadruples, quintuples, etc. We will see more ways to extract data from

tuples a little later.

Now an interesting function that produces pairs lists is zip . This function takes two lists and joins them in a list joining their elements in a pair. It is a really simple function but it has tons of uses. It is especially useful when we want to combine two lists in some way or go through two lists simultaneously. Here's a demo:

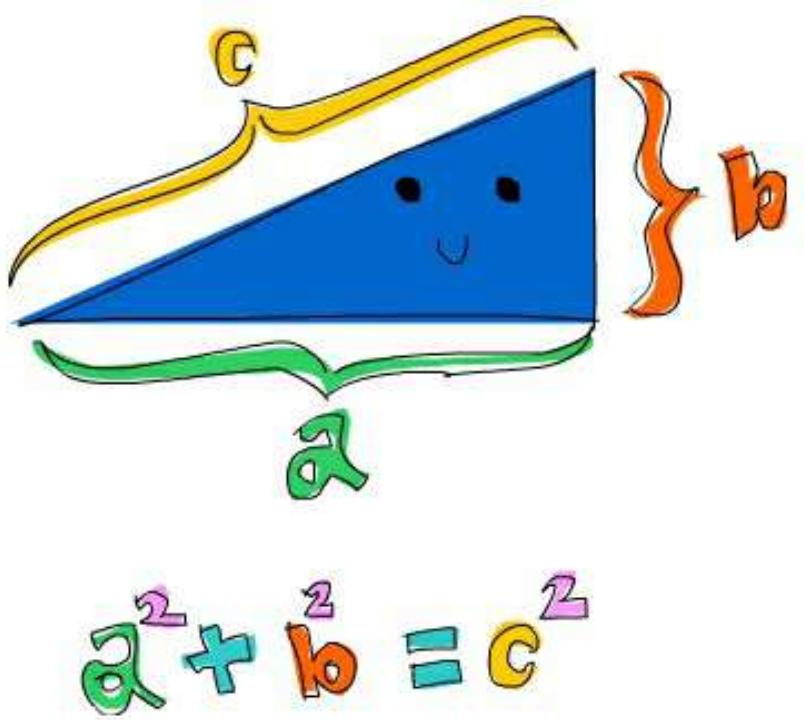
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1.5), (2.5), (3.5), (4.5), (5.5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1, "one"), (2, "two"), (3, "three"), (4, "four"), (5, "five")]
```

As we see, the elements are paired producing a new list. The first element goes the first, the second the second, etc. Note that since pairs can have different types, zip can take two lists containing different types and combine them. What if the size of the lists does not match?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["I am", "one", "turtle"]
[(5, "I am"), (3, "one"), (2, "turtle")]
```

The longest list is simply trimmed to match the size of the shortest. Since Haskell is lazy, we can use zip using finite and infinite lists:

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



Here's a problem that combines tuples with intensional lists: What right triangle whose sides measure integers less than 10 have a perimeter equal to 24? First, let's try to generate all triangles with sides equal to or less than 10:

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

We are simply extracting values from these three lists and our output function is combining them into a triple. If we avoid this by writing triangles in GHCi, we will get a list with all the possible triangles whose sides are less than or equal to 10. Now, we must add a condition that only filters out the right triangles. We are going to modify this function considering that side b is not longer than the hypotenuse and that side a is not longer than side b.

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

We're almost done. Now, we will simply modify the function saying that we only want those whose perimeter is 24.

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

And there is our answer! This method of troubleshooting is very common in functional programming. You start by taking a set of solutions and you apply transformations to get solutions, filtering them over and over again until you

get the correct solutions.

Chapter I

Types and type classes

Believe in the type

We mentioned earlier that Haskell has a static type system. The type of each expression is known at compile time, which produces more secure code. If we write a program that tries to divide a value of the Boolean type by a number, it will not compile. This is good as it is better to catch these kinds of errors at compile time rather than the program crashing. Everything in Haskell has a type, so the compiler can reason about the program before compiling it.

Unlike Java or C, Haskell has type inference. If we write a number, we don't have to specify that that's a number. Haskell can figure this out for himself, so we don't have to explicitly write the types of our functions or expressions to get results. We have already covered part of the Haskell foundations with very little knowledge of the types. However, understanding the type system is a very important part of mastering Haskell.

A type is like a label that has all expressions. This tag tells us what category the expression fits into. The expression `True` is a Boolean, `"Hello"` is a string, etc.

Now we are going to use GHCi to examine the types of some expressions. We will do it thanks to the command `:t` , which, followed by a valid expression tells us its type. Let's take a look:

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

```
(True, 'a') :: (Bool, Char)
ghci>: t 4 == 5
4 == 5 :: Bool
```

We can see that executing the command `: t` on an expression shows that same expression followed by `::` and its type. `::` can be read as *type* . Explicit types are always written with their first letter in capital letters. `'a'` , as we have seen, has the type `Char` . The name of this type comes from "Character". `True` has the `Bool` type . Makes sense. But what is this? Examining the type of `"HELLO!"` we get `[Char]` . The square brackets define a list. So we read this as a *list of characters* . Unlike lists, each tuple size has its own type. So the expression `(True, 'a')` has the type `(Bool, Char)` , while the expression `('a', 'b', 'c')` has the type `(Char, Char, Char)` . `4 == 5 will` always return `False` so this expression has type `Bool` .

Functions also have types. When we write our own functions we can give them an explicit type in their declaration. It is generally well considered to write types explicitly in the declaration of a function, except when they are very short. From now on we will give explicit types to all the functions we create. Remember the intensional list that filtered only the capitals of a string? Here's what it would look like with its type declaration:

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

`removeNonUppercase` has the type `[Char] -> [Char]` , which means it is a function that takes one string and returns another string. The `[Char]` type is synonymous with `String` so it would be more elegant to write the type as `removeNonUppercase :: String -> String` . Previously we didn't give this function a type since the compiler can infer it by itself. But how do we write the type of a function that takes multiple parameters? Here's a function that takes three integers and adds them together:

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

The parameters are separated by `->` and there is no special difference between the parameters and the type that the function returns. The type returned by the function is the last element of the declaration and the parameters are the rest. We'll see later why they are simply separated by `->` instead of having some kind of more explicit distinction between parameter types and return type, something like `Int, Int, Int -> Int` .

If we write a function and we are not clear about the type it should have, we can always write the function without its type and execute the command `:t` on it. Functions are also expressions so there is no problem using `:t` with them.

Here is a description of the most common types:

- `Int` represents integers. It is used to represent integers, so `7` can be an `Int` but `7.2` cannot. `Int` is bounded, which means it has a maximum value and a minimum value. Normally on 32-bit machines the maximum value of `Int` is 2147483647 and the minimum is -2147483648.
- `Integer` represents ... this ... integers too. The difference is that they are not bounded so they can represent very large numbers. However, `Int` is more efficient.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000(
```

- `Float` is a simple precision floating point real number.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
ghci> circumference 4.0
25.132742
```

- `Double` is a floating point real number of ... Double precision !.

```
circumference ':: Double -> Double
circumference 'r = 2 * pi * r
ghci> circumference '4.0
25.132741228718345
```

- `Bool` is the Boolean type. It can only have two values: `True` or `False` .
- `Char` represents a character. It is defined surrounded by single quotes. A character list is a string.

Tuples also have types but they depend on their length and the type of their components, so theoretically there are an infinity of types of tuples and that's too many types to cover in this guide. The empty tuple is also a type `()` which can only contain one value: `()` .

Type variables

What do you think is the type of the `head` function ? As `head it` takes a list of any type and returns its first element ... What could it be? Let's see it:

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

Hmmm ... what is `a` ? Is he a guy? If you remember before we said that types must start with capital letters, so it can't be exactly a type. Since it does not start with a capital letter it is actually a **type variable** . This means that `a` can be any type. It is similar to the generic types of other languages, only in Haskell they are much more powerful since it allows us to easily define very general functions as long as we do not make any specific use of the type in question. Functions that have type variables are called **polymorphic functions** . The `head` type declaration represents a function that takes a list of any type and returns an element of that same type.

Although type variables can have longer single-character names, we usually give them names like a, b, c, d, etc.

Do you remember `fst` ? Returns the first component of a pair. Let's examine its type.

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

As we see, `fst` takes a pair containing two types and returns an element of the same type as the first component of the pair. That is why we can use `fst` with pairs containing any combination of types. Note that just because `a` and `b` are different type variables they don't have to be different types. It simply represents that the first component and the value returned by the function are of the same type.

Type classes step by step (1st part)

Type classes are a kind of interface that defines some kind of behavior. If a type is a member of a type class, it means that type supports and implements the behavior that defines the type class. People who come from object-oriented languages are prone to confusing type classes because they think

they are like classes in object-oriented languages. Well, they are not. A more suitable approach would be to think that they are like Java interfaces, or Objective-C protocols, but better.

What is the type declaration of the `==` function ?

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

Note

The equality operator `==` is a function. So are `+` , `-` , `*` , `/` and almost all operators. If the name of a function is made up of only special characters (not alphanumeric), it is considered an infix function by default. If we want to examine its type, pass it to another function or call it in a prefix we must surround it with parentheses. For example: `(+) 1 4` equals `1 + 4` .

Interesting. Here we see something new, the symbol `=>` . Anything before the `=>` symbol is a class constraint. We can read the above type declaration as: the equality function takes two parameters that are of the same type and returns a `Bool` . The type of these two parameters must be a member of the `Eq` class (this is the class constraint).

The `Eq` type class provides an interface for equality comparisons. Any type that makes sense to compare two such values for equality must be a member of the `Eq` class . All Haskell standard types except IO type (a type to handle input / output) and functions are part of `Eq` class .

The `elem` function has the type `(Eq a) => a -> [a] -> Bool` because it uses `==` on the elements of the list to find out if the indicated element exists within the list.

Some basic type classes are:

- `Eq` is used by types that support equality comparisons. Members of this class implement the `==` or `/=` functions somewhere in their definition. All the types mentioned above are part of the `Eq` class except for the functions, so we can make equality comparisons on them.

- ghci> 5 == 5
- True
- ghci> 5 / = 5
- False
- ghci> 'a' == 'a'
- True

- ghci> "Ho Ho" == "Ho Ho"
- True
- ghci> 3,432 == 3,432
- True

- Ord is for guys who have some order.

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

All the types we have come to see except the functions are part of the Ord class . Ord covers all comparison functions like > , < , >= and <= . The compare function takes two members of the Ord class of the same type and returns their order. The order is represented by the Ordering type which can have three different values: GT , EQ and LT which represent *greater than* , *equal to* and *less than* , respectively.

To be a member of Ord , a guy must first be a member of the prestigious and exclusive Eq club .

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

- Show members can be represented by strings. All the types we have seen except the functions are part of Show . The most used function that works with this type class is the show function . It takes a value of a type that belongs to the Show class and represents it as a text string.

- ghci> show 3
- "3"
- ghci> show 5,334
- "5,334"
- ghci> show True
- "True"

- Read is like the opposite type class to Show . The read function takes a string and returns a value of the type that is a member of Read .

- ghci> read "True" || False
- True
- ghci> read "8.2" + 3.8

- 12.0
- ghci> read "5" - 2
- 3
- ghci> read "[1,2,3,4]" ++ [3]
- [1,2,3,4,3]

So far so good. Again, all the types we have seen except the functions are part of this type class. But what if we just use `read "4"` ?

```
ghci> read "4"
<interactive>: 1: 0:
    Ambiguous type variable `a 'in the constraint:
      `Read a 'arising from a use of` read' at <interactive>: 1: 0-7
    Probable fix: add a type signature that fixes these type variable (s)
```

What GHCi is not trying to say is that it does not know that we want it to return. Keep in mind that when we previously used `read` we did it by doing something later with the result. In this way, GHCi could infer the type of the result of the `read` function . If we use the result of applying the function as a boolean, Haskell knows that it has to return a boolean. But now, all he knows is that we want a type of the `Read` class , but not what. Let's take a look at the type declaration of the `read` function .

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

You see? Returns a type that is a member of the `Read` class , but if we don't use it elsewhere then, there is no way to know what type it is. For this reason we use explicit **type annotations** . Type annotations are a way of explicitly saying what type an expression should have. We do this by adding `::` to the end of the expression and then specifying the type. Observe:

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

Most expressions are of the type that the compiler can infer on its

own. But sometimes the compiler doesn't know the type of value an expression like `read "5"` should return , which could be `Int` , `Double` , etc. To find out, Haskell must actually evaluate `read "5"` . But since Haskell is a language with static types, it must know all types before the code is compiled (or in GHCi, evaluated). So with this we are saying to Haskell: "Hey, this expression should be of this type in case you don't know what it is."

- Members of the `Enum` class are sequentially ordered types, that is, they can be enumerated. The main advantage of the `Enum` type class is that we can use members in arithmetic lists. They also have successors and predecessors defined, so we can use the functions `succ` and `pred` . The types of this class are: `()` , `Bool` , `Char` , `Ordering` , `Int` , `Integer` , `Float` and `Double` .

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

- `Bounded` members have lower and upper limits, that is, they are bounded.

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

`minBound` and `maxBound` are interesting since they have the type `(Bounded a) => a` . That is, they are polymorphic constants.

All tuples are also `Bounded` if their components are too.

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

- `Num` is the class of numeric types. Its members have the property of being able to behave like numbers. Let's examine the type of a number.

```
ghci>: t 20
```

- 20 :: (Num t) => t

It seems that all the numbers are also polymorphic constants. They can act as if they were any type of the `Num` class .

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

These are the standard types of the `Num` class . If we examine the type of `*` we will see that it can accept any type of number.

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

Take two numbers of the same type and return a number of the same type. That is the reason why `(5 :: Int) * (6 :: Integer)` will throw an error while `5 * (6 :: Integer) will` work correctly and produce an `Interger` , since `5` can act as either an `Integer` or a `Int` .

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

- `Integral` is also a numeric type class. `Num` includes all numbers, including real and integer numbers. `Integral` only includes integers. `Int` and `Integer` are members of this class.
- `Floating` includes only floating point numbers, i.e. `Float` and `Double` .

Loading...

A very useful function for working with numbers is `fromIntegral` . It has the type `fromIntegral :: (Num b, Integral a) => a -> b` . From this statement we can say that it takes an integer and converts it to a more general number. This is useful when you are working with real and integers at the same time. For example, the `length` function has the type `length :: [a] -> Int` instead of having a more general type like `(Num b) => length :: [a] -> b` . I think it is for historical reasons or something similar, in my opinion, it is absurd. Anyway, if we want to get the size of a list and add `3.2` , we will get an error when trying to add an integer with one in a floating point. To solve this, we do `fromIntegral (length [1,2,3,4]) + 3.2` .

Note that in the fromIntegral type declaration there are several class restrictions. It is completely valid as you can see, the class restrictions must be separated by commas and in parentheses.

Chapter II

The syntax of functions

Pattern adjustment

This chapter will cover some of the most interesting syntactic constructions Haskell, starting with **setting patterns** (" *pattern matching* " in English). A pattern fit consists of a specification of patterns that must be followed by the data, which can be deconstructed allowing us to access its components.

We can separate the body that defines the behavior of a function into several parts, so that the code is much more elegant, clean and easy to read. We can use the adjustment of patterns with any type of data: numbers, characters, lists, tuples, etc. We are going to create a very trivial function that checks if the number we pass to it is a seven or not.

```
lucky :: ( Integral a ) => a -> String
lucky 7 = "The lucky seven!"
lucky x = "Sorry, it's not your lucky day!"
```

When we call lucky , the patterns are checked from top to bottom and when a pattern matches the associated value, the body of the associated function is used. In this case, the only way for a number to match the first pattern is for that number to be 7. If it is not, the next pattern will be evaluated, which matches any value and binds it to x . It could also have been implemented using an if statement . But, what if we wanted a function that named the numbers from 1 to 5, or "Not between one 1 and 5" for any other number? If we didn't have the pattern matching we should create a convoluted if then else tree . However with him:

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

Keep in mind that if we move the last pattern (the most general one) to the beginning, we would always get "Not between one 1 and 5" as an answer, since the first pattern would match any number and there would be no possibility of the other patterns being checked .

Do you remember the factorial function we created earlier? We define the factorial of a number n as product [1..n] . We can also implement a recursive factorial function, similar to how we would do it in mathematics. We start by saying that the factorial of 0 is 1. Then we say that the factorial of any other positive integer is that integer multiplied by the factorial of its predecessor.

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

This is the first time that we have defined a recursive function. Recursion is very important in Haskell, but we will talk about it later. In short, this is what happens when we try to get the factorial of, say 3. First try to calculate `3 * factorial 2`. The factorial of 2 is `2 * factorial 1`, so now we have `3 * (2 * factorial 1)`. `factorial 1` is `1 * factorial 0`, which brings us to `3 * (2 * (1 * factorial 0))`. Now comes the trick, we have defined the factorial of 0 to be just 1, and since it meets that pattern before the other more general one we get 1. So the result equals `3 * (2 * (1 * 1))`. If we had written the second pattern at the beginning, it would have accepted all the numbers including 0 and the calculation would never finish. For this reason the order is important when defining the patterns and it is always better to define the most specific patterns at the beginning leaving the most general ones at the end.

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

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

And we try to run it with an unexpected value, this is what happens:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs: (53.0) - (55.21): Non-exhaustive patterns in function charName
```

It complains that we have a non-exhaustive pattern matching and it certainly is. When we use patterns we always have to include a general one to make sure that our program will not fail.

Pattern matching can also be used with tuples. How would we create a function that would take two 2D vectors (represented with pairs) and return the sum of both? To add two vectors we first add their `x` components and their `y` components separately. This is how we would do it if there was no pattern matching:

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

Okay, it works, but there are better ways to do it. We are going to modify the function to use a pattern fit.

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

There you have it! Better. Keep in mind that it is a general pattern, that is, it

will be verified for any pair. The type of `addVectors` is in both cases the same: `addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)` , so it is guaranteed that we will have two pairs as parameters.

`fst` and `snd` extract components from the pairs. But what about the triples? Well, since we don't have functions that do the same with triples, we are going to create them ourselves.

```
first :: ( a , b , c ) -> a
first ( x , _ , _ ) = x

second :: ( a , b , c ) -> b
second ( _ , y , _ ) = y

third :: ( a , b , c ) -> c
third ( _ , _ , z ) = z
```

`_` has the same meaning as with intensional lists. Denotes that we do not really care about that value, since we will not use it.

We can also use pattern matching with intensional lists. Notice:

```
ghci> let xs = [(1.3), (4.3), (2.4), (5.3), (5.6), (3.1)]
ghci> [a + b | (a, b) <- xs]
[4,7,6,8,11,4]
```

In the event of a pattern failure, it will simply go to the next element.

Lists can also be used in a pattern fit. You can compare against the empty list `[]` or against any pattern that involves `:` and the empty list. Like `[1,2,3]` , which is just another way of expressing `1: 2: 3: []` (we can use both alternatives). A pattern like `x: xs` will bind the head of the list with `x` and the rest with `xs` , even when the list has only one element, in which case `xs` will end up being the empty list.

Note

The `x: xs` pattern is widely used, especially with recursive functions. Patterns containing a `:` will only accept lists with some element.

If we wanted to link, say, the first three elements of a list to variables and the rest to another variable we can use something like `x: y: z: zs` . However this will only accept lists that have at least 3 elements.

Now that we know how to use patterns with lists, let's implement our own

head function .

```
head ' :: [ a ] -> a
head ' [] = error "Hey, you can't use head with an empty list!"
head ' ( x : _ ) = x
```

We check that it works:

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

Well! Note that if we want to link several variables (even if some of them are _ and we don't really want to link them) we must surround them with parentheses. Also look at the error function we just used. It takes a string and generates a runtime error using the string that we pass to it as information about the error that occurred. It causes the program to end, which is not good to use very often. Anyway, calling head with an empty list doesn't make much sense.

We are going to create a function that tells us some of the first elements that a list contains.

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

This function is safe as it takes into account the possibility of an empty list, a list with one element, a list with two elements and a list with more than two elements. Note that we could write (x: []) y (x: y: []) as [x] and [x, y] without using parentheses. But we can't write (x: y: _) using square brackets since it accepts lists with more than two elements.

We already implemented the length function using intensional lists. Now we are going to implement it with a pinch of recursion.

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

It is similar to the factorial function that we wrote before. We first define the result of a known entry, the empty list. This is also known as the base case. Then in the second pattern we divide the list on its head and the rest. We say

the length is 1 plus the size of the rest of the list. We use `_` for the head of the list since we are not really interested in its content. Note that we have also taken into account all possible cases of lists. The first pattern accepts the empty list, and the second accepts all others.

Let's see what happens if we call `length '` with `"eye"` . First it would be checked if it is an empty list, as it is not, we would continue to the next pattern. This is accepted and tells us that the length is `1 + length ' "jo"` , since we have divided the string into head and tail, decapitating the list. Voucher. The size of `"jo"` is similarly `1 + length ' "or"` . So right now we have `1 + (1 + length ' "o")` . `length ' "or"` is `1 + length' ""` (we could also write it as `1 + length ' []`). And since we have `length ' []` `set` to 0, at the end we have `1 + (1 + (1 + 0))` .

Now we will implement `sum` . We know that the sum of an empty list is 0, which we write with a pattern. We also know that the sum of a list is the head plus the sum of the rest of the tail, and if we write it we obtain:

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

There are also so-called *patterns* , or *as patterns* (from English, *as patterns*). They are useful to decompose something using a pattern, so that it is linked with the variables that we want and also we can keep a reference to that something as a whole. For this we put an `@ in` front of the pattern. The best way to understand it is with an example: `xs @ (x: y: ys)` . This pattern will fit exactly the same as `x: y: ys` but we could also easily access the entire list using `xs` instead of having to repeat ourselves typing `x: y: ys` in the body of the function. A quick example:

```
capital :: String -> String
capital "" = "An empty string!"
capital all @ ( x : _ ) = "The first letter of" ++ all ++ "is" ++ [ x ]
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

Normally we use *patterns* to avoid repeating when we are adjusting a larger pattern and we have to use it whole again somewhere in the body of the function.

One more thing, we can't use `++` in pattern settings. If we try to use a pattern `(xs ++ ys)` , what would be in the first list and what in the second? Dont have

much sense. It would make more sense to adjust patterns like `(xs ++ [x, y, z])` or just `(xs ++ [x])` but given the nature of the lists we can't do this.

Guardians, Guardians!

While patterns are a way of making sure a value has a certain shape and deconstructing it, guards are a way of checking if any property of a value (or several of them) is true or false. It sounds very much like an `if statement` and is actually very similar. The thing is, the guards are much more readable when you have various conditions and fit the patterns very well.

Instead of explaining its syntax, we are simply going to create a function that uses guards. We will create a simple function that will scold you differently based on your BMI (body mass index). Your BMI equals your height divided by your weight squared. If your BMI is less than 18.5 you are underweight. If you are somewhere between 18.5 and 25 you are out of the bunch. If you are between 25 and 30 you are overweight and if you are over 30 you are obese. So here is the function (we are not calculating anything now, just get a BMI and scold you)

```
bmiTell :: ( RealFloat a ) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You are underweight, are you emo?"
    | bmi <= 25.0 = "Supposedly you're normal ... I hope you're ugly."
    | bmi <= 30.0 = "You're fat! You lose some chubby weight."
    | otherwise = "Congratulations, you are a whale!"
```

The guards are indicated by vertical bars that follow the function name and its parameters. They are usually indented and aligned. A guard is basically a boolean expression. If `True` is evaluated , then the corresponding function body is used. If evaluated to `False` , the next save is checked, and so on. If we call this function with `24.3` , it will first check if it is less than or equal to `18.5` . As it is not, it will continue to the next guard. The second guard is checked and since 24.3 is less than 25, the second string is returned.

It is reminiscent of a large `if then else` tree of imperative languages, only much clearer. Generally very large `if else` trees are frowned upon, but there are times when a problem is discretely defined and there is no way to fix it. The guards are a good alternative for this.

Many times the last guard is `otherwise` . `otherwise` is simply defined as `otherwise = True` and accepts everything. It is very similar to pattern matching, they are only accepted if the input satisfies a pattern, but the guards check boolean conditions. If all the saves of a function are evaluated to `False` (and we have not given another save `otherwise`), the evaluation fails and will continue to the next **pattern** . This is why patterns and guards fit so well together. If there is no acceptable pattern or guard, an error will be thrown.

Of course we can use guards with functions that take as many parameters as you want. Instead of letting the user calculate their own BMI on their own before calling the function, we are going to modify the function so that it takes the height and weight and calculates it for us.

```
bmiTell :: ( RealFloat a ) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You are underweight. Are you emo?"
    | weight / height ^ 2 <= 25.0 = "Supposedly you're normal ... I hope you're ugly."
    | weight / height ^ 2 <= 30.0 = "You're fat! You lose some chubby weight."
    | otherwise = "Congratulations, you are a whale!"
```

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

```
ghci> bmiTell 85 1.90
"You're supposed to be normal ... I hope you're ugly."
```

Yes! I'm not fat, but Haskell just called me ugly ...

Note that there is no `=` after the function name and its parameters, before the first save. Many newbies get a syntax error by putting an `=` there, and you will too.

Another very simple example: we are going to implement our `max` function . If you remember, it can take two things that can be compared and it returns the highest.

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

The guards can also be written in a single line, although I warn that it is better not to do it since they are much less readable, even with short functions. But to demonstrate it we can define `max'` as:

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

Arg! It is not easily read. Let's move on. We are going to implement our own `compare` using guards.

```
myCompare :: ( Ord a ) => a -> a -> Ordering
to ` myCompare ` b
    | a > b = GT
    | a == b = EQ
    | otherwise = LT
ghci> 3 `myCompare` 2
GT
```

Note

Not only can we call functions infixed using quotation marks, but we can also define them in this way. Sometimes it is easier to read it like this.

Where?

In the previous section we defined the function that calculated the BMI like this:

```
bmiTell :: ( RealFloat a ) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You are underweight. Are you emo?"
    | weight / height ^ 2 <= 25.0 = "Supposedly you're normal ... I hope you're ugly."
    | weight / height ^ 2 <= 30.0 = "You're fat! You lose some chubby weight."
    | otherwise = "Congratulations, you are a whale!"
```

If you look you will notice that we repeat ourselves three times. We repeat three times. Repetition (three times) while programming is as desirable as being kicked where it hurts the most. Since we are repeating the same expression three times it would be ideal if we could compute it once, bind it to a variable, and use it instead of the expression. Well, we can modify our function like this:

```
bmiTell :: ( RealFloat a ) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You are underweight, are you emo?"
    | bmi <= 25.0 = "Supposedly you're normal ... I hope you're ugly."
    | bmi <= 30.0 = "You're fat! You lose some chubby weight."
    | otherwise = "Congratulations, you are a whale!"
    where bmi = weight / height ^ 2
```

We have put the `where` keyword after the guards (it is usually best to align it with the rest of the vertical bars) and then define several variables. These variables are visible on the guards and give us the advantage of not having to repeat ourselves. If we decide that we have to calculate the BMI in another way, we only have to modify it in one place. It also improves readability as it names things and makes our programs faster since things like `bmi` only need to be calculated once. We could go a bit and present a function like this:

```
bmiTell :: ( RealFloat a ) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You are underweight, are you emo?"
    | bmi <= normal = "Supposedly you're normal ... I hope you're ugly."
    | bmi <= fat = "You're fat! Lose some chubby weight."
    | otherwise = "Congratulations, you are a whale!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

The variables that we define in the where section of a function are only visible from that function, so we don't have to worry about them when creating more variables in other functions. If we don't align the where section well and correctly Haskell will get confused because he won't know which group he belongs to.

Variables defined with where are not shared between the bodies of different patterns of a function. If we want several patterns to access the same variable, we must define it globally.

We can also use pattern matching with the where sections . We could rewrite the where section of our previous function as:

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

We are going to create another trivial function in which given a first and last name returns its initials.

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

We could have done the pattern matching directly on the function parameters (actually it would have been shorter and fancier) but so we can see what is possible with the where sections .

In the same way that we have defined constants in blocks where we can also define functions. Staying true to our health program we are going to do a function that takes a list of weight and height pairs and returns a list of IMCs.

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

There you have it! The reason we have created the bmi function in this

example is that we cannot simply calculate a BMI from the parameters of our function. We have to examine all the elements of the list and calculate their BMI for each pair.

Where sections can also be nested. It is very common to create a function and define some auxiliary functions in the where section and then define other auxiliary functions within each of them.

Let it be

Very similar to where sections are let expressions . Where sections are a syntactic construct that let you bind variables to the end of a function so that the entire function can access it, including all guards. Let expressions are used to bind variables anywhere and are expressions in themselves, but they are very local, so they cannot be extended between guards. Like all Haskell constructs that allow you to bind values to variables, let expressions allow you to use pattern matching. Let's see it in action! This is how we could define a function that would give us the area of a cylinder based on its height and its radius.

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

Its form is let <definition> in <expression> . The variables that we define in the let expression are accessible in the in part . As we can see we could also have defined this with a where section . Also note that the names are aligned in the same column. So what is the difference between them? For now it seems that let puts the definitions first and then the expression that uses them while where does it in the reverse order.

The difference is that let expressions are expressions by themselves. Where sections are simply syntactic constructions. Remember when we explained if statements and explained that since they are an expression they can be used almost anywhere?

```
ghci> [if 5> 3 then "Woo" else "Boo", if 'a'> 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
```

```
ghci> 4 * (if 10> 5 then 10 else 0) + 2
42
```

You can also do the same with `let` expressions .

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

They can also be used to define functions at a local level:

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

If we want to link multiple variables on a single line, obviously we can't align the definitions in the same column. For this reason we can separate them with semicolons.

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

We don't have to put the last semicolon but we can do it if we want. As we have already said, we can use pattern adjustments with `let` expressions . They are very useful to dismantle tuples into their components and link them to various variables.

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

We can also use `let` sections within intensional lists. Let's rewrite our previous example that computed a list of height and weight pairs to use a `let` within an intensional list instead of defining an auxiliary function with a `where` .

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

We include a `let` inside the intensional list as if it were a predicate, only it doesn't filter the elements, it only binds variables. Variables defined in a `let` expression within an intensional list are visible from the output function (the part before `|`) and all the predicates and sections that come after their definition. We could make our function return BMI only for obese people like so:

```
calcBmis :: ( RealFloat a ) => [( a , a )] -> [ a ]
```

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

We cannot use the name bmi inside the (w, h) <- xs part since it is defined before the let expression .

We omit the in part of the let sections within the intensional lists because the visibility of names is predefined in these cases. However, we can use a let in section in a predicate and the defined variables will only be visible in this predicate. The in part can also be omitted when we define functions and constants inside the GHCi interpreter . If we do, the variables will be visible throughout the session.

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

If let expressions are so interesting, why not always use them instead of where sections ? Well, since let expressions are expressions and are quite local in scope, they cannot be used between guards. There are people who prefer where sections because variables come after the function that uses them. In this way, the body of the function is closer to its name and type declaration and some think it is more readable.

Case expressions

Many imperative languages (such as C, C ++, Java, etc.) have case syntax constructs and if you've ever programmed into them, you probably know what this is about. It's about taking a variable and then executing code blocks for certain specific values of that variable and then maybe including some block that always runs in case the variable has some value that doesn't match any of the above.

Haskell takes this concept and takes it one step further. As the name implies, case expressions are, well, expressions, like if else expressions or let expressions . Not only can we evaluate expressions based on the possible

values of a variable, but we can perform a pattern fit. Mmmm ... take a value, do a pattern matching on it, evaluate chunks of code based on its value, where have we heard this before? Oh yes, in the pattern settings of a function's parameters. Well, it is actually a syntactic alternative to case expressions . These two pieces of code do the same thing and are interchangeable:

```
head ' :: [ a ] -> a
head ' [] = error "head does not work with empty lists!"
head ' ( x : _ ) = x
head ' :: [ a ] -> a
head ' xs = case xs of [] -> error "head does not work on empty lists!"
                       ( x : _ ) -> x
```

As you can see the syntax for case expressions is very simple.

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

The expression is adjusted against the patterns. The pattern matching action behaves as expected: the first pattern to match is the one used. If it cannot be adjusted to any pattern of the case expression it will throw an execution error.

While pattern matching of function parameters can only be done when defining a function, case expressions can be used almost anywhere. For example:

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

Loading...

They are useful for adjusting patterns in the middle of an expression. Since the pattern matching performed in the definition of a function is a syntactic alternative to case expressions , we could also use something like this:

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

Chapter III

Recursion

Hello recursion!

In the previous chapter we already mentioned recursion. In this chapter we will take a closer look at this topic, why it is important in Haskell and how we can create solutions to problems in an elegant and concise way.

If you still don't know what recursion is, read this sentence: Recursion is actually a way of defining functions in which this function is used in the definition of the function itself. Mathematical definitions are usually defined recursively. For example, the Fibonacci series is defined recursively. First, we define the first two Fibonacci numbers non-recursively. We say that F (0) = 0 and F (1) = 1 , which means that the 1st and 2nd Fibonacci numbers are 0 and 1, respectively. Then for any other index, the Fibonacci number is the sum of the two previous Fibonacci numbers. So F (n) = F (n-1) + F (n-2) . Thus, F (3) = F (2) + F (1) which is F (3) = (F (1) + F (0)) + F (1) . Since we have gone down to the only non-recursively defined numbers in the Fibonacci series, we can assure that F (3) = 2 . Non-recursively defined elements, such as F (0) or F (1) , are called **base cases** , and if we have only base cases in a definition such as F (3) = (F (1) + F (0)) + F (1) is called **a boundary condition** , which is very important if you want your function to end. If we had not defined F (0) and F (1) non-recursively, we would never obtain a result for any number, since we would reach 0 and continue with the negative numbers. Suddenly, we would find an F (-2000) = F (-2001) + F (-2002) and we would still not see the end.

Recursion is very important in Haskell since, unlike in imperative languages, we perform calculations declaring how something **is** , rather than declaring **how to** get something. For this reason there are no while loops or for loops in Haskell and instead we have to use recursion to declare what something is.

The impressive maximum

The maximum function takes a list of things that can be ordered (that is, instances of the Ord type class) and returns the largest. Think about how we would implement this imperatively. We would probably create a variable to hold the maximum value so far and then loop through the items in the list so that if an item is greater than the current maximum value, we would replace it. The maximum value that is kept at the end is the result. Wow! there are many words to define such a simple algorithm.

Now let's see how we would define this recursively. We could first establish a base case by saying that the maximum of a unit list is the only element that contains the list. Then we could say that the maximum of a longer list is the head of that list if it is greater than the maximum of the queue, or the maximum of the queue if it is not. That's! We are going to implement it in Haskell.

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

As you can see the pattern matching works great in conjunction with recursion. Many imperative languages do not have patterns, so many if / else must be used to implement base cases. The first base case says that if a list is empty, Error! It makes sense because what is the maximum of an empty list? No idea. The second pattern also represents a base case. It says that if they give us a unit list we simply return the single element.

The third pattern is where the action is. We use a pattern to divide the list into head and tail. This is very common when we use a recursion with lists, so get used to it. We use a where section to define maxTail as the maximum of the rest of the list. Then we check if the head is larger than the rest of the tail. If it is, we return the head, if not, the maximum of the rest of the list.

Let's take a list of example numbers and see how it works: [2,5,1] . If we call maximum ' with this list, the first two patterns would not fit. The third would

do it and the list would be divided into 2 and [5,1] . The where section requires knowing the maximum of [5.1] so we go there. It would fit the third pattern again and [5,1] would be divided into 5 and [1] . Again, the where section requires knowing the maximum of [1] . Since this is a base case, it returns 1 at last! So we go up one step, we compare 5 with the maximum of [1] (which is 1) and surprisingly we get 5. So now we know that the maximum of [5.1] is 5 . We go up another step and we have 2 and [5,1] . We compare 2 with the maximum of [5.1] , which is 5 and choose 5 .

A clearer way of writing the maximum ' function is using the max function . If you remember, the max function takes two things that can be ordered and returns the largest of them. This is how we could rewrite the function using max :

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

What is elegant? In short, the maximum of a list is the maximum between its first element and the maximum of the rest of its elements.

A few more recursive functions

Now that we know how to think recursively in general, let's implement a few functions recursively. First, we are going to implement replicate . replicate takes an Int and some element and returns a list containing several repetitions of that same element. For example, replicate 3 5 returns [5.5,5] . Let's think about the base case. My intuition tells me that the base case is 0 or less. If we try to replicate something 0 or less times, we must return an empty list. Also for negative numbers since it doesn't make sense.

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

Here we use guards instead of patterns because we are checking a boolean condition. If n is less than or equal to 0 we return an empty list. Otherwise we

return a list that has x as its first element and x replicated n-1 times as its tail. Finally, part n-1 will make our function reach the base case.

Now we are going to implement take . This function takes a certain number of elements from a list. For example, take 3 [5,4,3,2,1] will return [5,4,3] . If we try to get 0 or less elements from a list, we will get an empty list. Also if we try to take something from an empty list, we will get an empty list. Note that both are base cases. Let's write it down.

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

The first pattern indicates that if we want to get 0 or a negative number of elements, we get an empty list. Notice that we are using _ to link the list since we don't really care in this pattern. In addition we are also using a guard, but without the otherwise part . This means that if n ends up being something more than 0, the pattern will fail and continue to the next one. The second pattern indicates that if we try to take something from an empty list, we get an empty list. The third pattern breaks the list at the head and tail. Then we say that if we take n elements from a list it is equal to a list that has x as head and as a tail a list that takes n-1 elements from the queue. Try using paper and pencil to follow the development of what the take 3 [4,3,2,1] evaluation would be , for example.

reverse simply reverses a list. Think about the base case, what is it? Let's see ... It's an empty list! A reverse empty list equals that same empty list. Okay, how about the rest of the list? We could say that if we divide a list into its head and tail, the reverse list equals the inverted tail plus then the head at the end.

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

There you have it!

Since Haskell supports infinite lists, our recursion doesn't actually have to

have base cases. But if it does not have them, we will continue calculating something infinitely or producing an infinite structure. However, the good thing about these infinite lists is that we can cut them wherever we want. repeat takes an element and returns an infinite list that simply has that element. An extremely simple recursive implementation is:

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

Calling repeat 3 would give us a list that has a 3 in its head and then it would have an infinite list of three's in its tail. So repeat 3 would evaluate to something like 3: (repeat 3) , which is 3: (3: (repeat 3)) , which is 3: (3: (3: (repeat 3))) , etc. repeat 3 will never finish its evaluation, while take 5 (repeat 3) will return a list with five threees. It is the same as doing replicate 5 3 .

zip takes two lists and combines them into one. zip [1,2,3] [2,3] returns [(1,2), (2,3)] as it truncates the longest list to match the shortest. What happens if we combine something with the empty list? Well, we would get an empty list. So this is our base case. However, zip takes two lists as parameters, so we actually have two base cases.

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

The first two patterns say that if the first or second list is empty then we get an empty list. Combining [1,2,3] and ['a', 'b'] will end by trying to combine [3] and [] . The base case will appear on the scene and the result will be (1, 'a') :(2, 'b'): [] that exactly the same as [(1, 'a'), (2, 'b')] .

We are going to implement one more function from the standard library, elem , which takes an element and a list and searches if that element is in that list. The base case, like most of the time with lists, is the empty list. We know that an empty list does not contain elements, so it most likely does not contain the element we are looking for ...

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

```
    | a == x = True
    | otherwise = a `elem` xs
```

Fairly simple and predictable. If the head is not an element that we are looking for then we look in the tail. If we get to an empty list, the result is false.

Quicksort!

We have a list of items that can be ordered. Its type is a member of the Ord type class . And now, we want to order them. There is a very interesting algorithm for ordering them called Quicksort. It is a very smart way to sort items. While in some imperative languages it can take up to 10 lines of code to implement Quicksort, in Haskell the implementation is much shorter and more elegant. Quicksort has become a kind of Haskell sample piece. Therefore, we are going to implement it, despite the fact that the implementation of Quicksort in Haskell is considered very cheesy since everyone does it in the presentations so we can see how nice it is.

Well, the type declaration will be quicksort :: (Ord a) => [a] -> [a] . No surprise. Base case? The list is empty, as expected. Now comes the main algorithm: an ordered list is a list that has all the elements less (or equal) than the head at the beginning (and those values are ordered), then comes the head of the list that will be in the middle, and then they come elements that are larger than the head (which will also be ordered). We've said "sorted" twice, so we'll probably have to make two recursive calls. We have also used the verb "is" twice to define the algorithm instead of "does this", "does that", "then does" ... That is the beauty of functional programming! How are we going to filter the elements that are greater and less than the head of the list? With intensional lists. So let's start and define this function:

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

```
        biggerSorted = quicksort [ a | a <- xs , a > x ]
    in smallerSorted ++ [ x ] ++ biggerSorted
```

We are going to run a little test to see if it behaves correctly.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the swift hindu bat ate happy cardillo and kiwi"
"aaaccddeeeefghiiiiiiiklllllmmnooooorruuvwyzz"
```

Well this is what we were talking about! So if we have, say [5,1,9,4,6,7,3] and want to sort them, the algorithm will first take the head of the list, which is 5 and put it in the middle of two lists that are the least and the older ones of this. In this way we will have (quicksort [1,4,3]) ++ [5] ++ (quicksort [9,6,7]) . We know that when the list is completely ordered, the number 5 will remain in the fourth position since there are three numbers less than and three numbers greater than it. Now if we order [1,4,3] and [9,6,7] , we will have an ordered list! We order these two lists using the same function. In the end we will reach a point where we will reach empty lists and empty lists are already sorted in some way. Here's an illustration:

An item that is in its correct position and will not move anymore is orange. Reading these items from left to right the list a appears ordered. Although we chose to compare all the elements with the head, we could have chosen any other element. In Quicksort, the element with which we compare is called pivot. These are the green ones. We chose the head because it is very easy to apply a pattern to it. Items that are smaller than the pivot are light green and items that are larger in black. The yellow gradient represents the Quicksort application.

Thinking recursively

We have used recursion a bit and as you may have noticed there are some common steps. Normally we first define base cases and then we define a function that does something between one element and the function applied to the other elements. It doesn't matter if this element is a list, a tree, or any other data structure. A summation is the sum of the first element plus the sum

of the rest of the elements. A producer is the product of the first element among the product of the rest of the elements. The size of a list is 1 plus the size of the rest of the list, etc.

Of course there are also base cases. Usually a base case is a scenario where applying a recursion is meaningless. When working with lists, base cases often deal with empty lists. When we use trees, the base cases are normally the nodes that do not have children.

It is similar when dealing with numbers. We usually do something with a number and then apply the function to that modified number. We have already done recursive functions of this type as the factorial of a number, which does not make sense with zero, since the factorial is only defined for positive integers. Often the base case turns out to be identity. The identity of the multiplication is 1 since if you multiply something by 1 you get the same result. Also when we do list summations, we define the sum of an empty list as 0, since 0 is the identity of the sum. In Quicksort, the base case is the empty list and the identity is also the empty list, because if you add the empty list to a list, you get the same ordered list.

When we want to solve a problem recursively, we first think where a recursive solution does not apply and if we can use this as a base case. Then we think about identities, where we should break the parameters (for example, the lists break at the head and tail) and where we should apply the recursive function.

Chapter IV

Higher order functions

Haskell functions can take functions as parameters and return functions as a result. A function that does both or both does is called a higher order function. Higher order functions are not just another part of Haskell, they represent the experience of programming in Haskell themselves. They appear when you want to define calculations by defining things as they are instead of defining the change steps of some state or some loop, higher order functions are indispensable. They are really a very powerful way of solving problems and thinking about programs.

Currified functions

Officially each Haskell function can only take one parameter. So how is it possible that we have defined and used various functions that take more than one parameter? Well it's a good trick! All the functions we have used so far and accepted more than one parameter have been curved functions. What does this mean? You will understand it better with an example. We are going to use our good friend, the max function . It seems to take two parameters and return the one that is greater. When applying max 4 5 First a function is created that takes a single parameter and returns 4 or the parameter, depending on which is greater. Then 5 is applied to that function and it produces the desired result. This sounds a bit complicated but it is actually a very useful concept. The following two calls are equivalent:

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

Putting a space between two things is simply **applying a function** . Space is a kind of operator and has the highest order of preference. Let's examine the max type . It is max :: (Ord a) => a -> a -> a . This can also be written as max :: (Ord a) => a -> (a -> a) . And it can also be read as: max takes an a and

returns (that is ->) a function that takes an a and returns an a . That is why the returned type and the function parameters are separated only by arrows.

And how does this benefit us? In short, if we call a function with parameters of less we get a **partially applied** function , that is, a function that takes as many parameters as it lacks. Using the partial function application (or calling functions with fewer parameters) is an easy way to create functions on the fly so that we can pass them as parameters to other functions or provide them with some data.

Take a look at this offensively simple feature.

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

What really happens when we perform multThree 3 5 9 or ((multThree 3) 5) 9 ? First, 3 is applied to multThree since it is separated by a space. This creates a function that takes a parameter and returns a function. Then 5 is applied to is, so a function will be created that takes a parameter and multiplies it by 15. 9 is applied to that function and the result is 135 or something similar. Remember that the type of this function could also be written as multThree :: (Num a) => a -> (a -> (a -> a)) . What is before the -> is the parameter that the function takes and what is after it is what it returns. So our function takes an a and returns a function with a type (Num a) => a -> (a -> a) . Similarly, this function takes an a and returns a function of the type (Num a) => a -> a . And finally, this function takes an a and returns an a . Watch this:

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

By calling functions with fewer parameters than necessary, clearly speaking, we create functions on the fly. What if we want to create a function that takes a number and compares it to 100? We could do something like this:

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

If we call it with 99 it returns GT . Pretty simple. Look at the x on the right

side of the equation. Now we are going to think that it returns `` compare 100 . Returns a function that takes a number and compares it to 100. Wow! Isn't that what we were looking for? We can rewrite it as:

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

The type declaration remains the same since compare 100 returns a function. compare has the type (Ord a) => a -> (a -> Ordering) and calling it with 100 returns (Num a, Ord a) => a -> Ordering . The additional class constraint is added because 100 is also part of the Num type class .

Note

Make sure you really know how curved functions and partial application of functions work as they are very important!

Infix functions can also be partially applied using sections. To section an infix function, simply surround it with parentheses and supply a single parameter on one side. This creates a function that takes a parameter and applies it to the missing side of an operand. An extremely trivial function would be:

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

Calling, say, divideByTen 200 is equivalent to making 200/10 or (/ 10) 200 . A function that checks if a character is uppercase would be:

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

The only special thing about the sections is the use of - . By definition, (-4) would be a function that takes a number and subtracts it from 4. However, for convenience, (-4) means minus four. So if you want a function that subtracts 4 from a number you can use (subtract 4) or ((-) 4) .

What if we try to multthree 3 4`` in GHCi instead of giving it a name with a `` let or passing it to another function?

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

In a 'do' expression: print it

GHCi is telling us that the produced expression is a function of type a -> a but does not know how to display it on the screen. Functions are not members of the Show type class , so we cannot get a string with the representation of a function. If we do something like 1 + 1 on GHCi, first compute that is 2 , and then call show at 2 to have a textual representation of that number. And a textual representation of 2 is simply "2" , which is what we get on screen.

Higher order in your order

Functions can take functions as parameters and also return functions. To illustrate this we are going to create a function that takes a function and applies it twice to something.

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

First look at its type declaration. Before, we didn't need to use parentheses since -> is naturally right associative. However, here is the exception. This indicates that the first parameter is a function that takes something and returns something of the same type. The second parameter is something of that same type and also returns something of that type. We could also read this type declaration curribly, but to save us from a good headache we will simply say that this function takes two parameters and returns only one thing. The first parameter is a function (of type a -> a) and the second is of the same type a . The function can be of type Int -> Int or of type String -> String or anything else. But then, the second parameter must be of the same type.

Note

From now on we will say that a function takes several parameters instead of saying that actually a function takes one parameter and returns a partially applied function until it reaches a function that returns a solid value. So for simplicity we will say that a -> a -> a takes two parameters, even though we know what is really going on.

The body of the function is very simple. We use the parameter f as a function,

applying x to it by separating them with a space, and then applying the result to f again. Anyway, play around with the function:

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

The incredible and usefulness of the partial application is evident. If our function requires us to pass it a function that takes a single parameter, we can simply partially apply a function to that which takes a single parameter and then pass it.

Now we are going to use higher order programming to implement a useful function that is in the standard library. It is called zipWith . It takes a function and two lists and joins them by applying the function between the corresponding parameters. Here is how we would implement it:

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

Look at the type statement. The first element is a function that takes two things and produces a third. They do not have to be the same type, although they can be. The second and third parameters are lists. The first has to be a list of a since the union function takes a as the first parameter. The second is a list of b . The result is a list of c . If the type declaration of a function says that it accepts a function a -> b -> c as a parameter, it will also accept a function of type a -> a -> a . Remember that when you are creating a function, especially a higher order one, and you are not sure of its type, you can simply omit the type declaration and then look at the type Haskell infers using : t .

The action of the function is very similar to that of zip . The base case is the same, only that there is an extra parameter, the join function, but this parameter doesn't matter in the base case so we use _ with it. The body of the

function for the last pattern is also very similar to that of zip , only it doesn't do (x, y) but f x y . A single higher order function can be used to perform a multitude of different tasks if it is general enough. Here's a little taste of what zipWith ' can do :

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

As you can see, a single higher order function can be used in a very versatile way. Imperative languages normally use things like while loops , setting some variable, checking their status, etc. to get similar behavior and then wrap it with an interface, a function. Functional programming uses higher-order functions to abstract common patterns, such as examining two lists in pairs and doing something with those pairs, or taking a set of solutions and removing those you don't need.

We are going to implement another function that is already in the standard library called flip . flip takes a function and returns a function that is like our original function, only the first two parameters are swapped. We can implement it like this:

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

Here, we take advantage of the fact that the functions are curred. When we call flip ' without the x and y parameters , it will return a function that takes those parameters but will call them backwards. Even though the functions to which flip has been applied are normally passed to other functions, we can take advantage of currification when we create higher order functions by thinking beforehand and writing their final result as if they were fully applied calls.

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

```
[5,4,3,2,1]
```

Associations and filters

map takes a function and a list and applies that function to each item in that list, producing a new list. Let's see its type definition and how it is defined.

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

The type definition says that it takes a function and that in turn it takes an a and returns a b , a list of a and returns a list of b . It is interesting that simply by looking at the type definition of a function, we can sometimes say what the function does. map is one of those higher order functions that are really versatile and can be used in millions of different ways. Here you have it in action:

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

You have probably noticed each of these statements can be achieved using comprehension lists. map (+3) [1,5,3,1,6] is the same as writing [x + 3 | x <- [1,5,3,1,6]] . However using map is much more readable when you only have to apply a function to the elements of a list, especially when you are dealing with mapping mappings so that it fills everything with a lot of square brackets and ends up being a mess.

filter is a function that takes a predicate (a predicate is a function that tells whether something is true or false, or in our case, a function that returns a boolean value) and a list and returns a list of the elements that satisfy the predicate . The type declaration and implementation would be something like:

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

```
filter _ [] = []
filter p ( x : xs )
    | p x = x : filter p xs
    | otherwise = filter p xs
```

Pretty simple. If p x evaluates to True then the element is included in the new list. If not, it stays out. Some examples:

```
ghci > filter ( > 3 ) [ 1 , 5 , 3 , 2 , 1 , 6 , 4 , 3 , 2 , 1 ]
[ 5 , 6 , 4 ]
ghci > filter ( == 3 ) [ 1 , 2 , 3 , 4 , 5 ]
[ 3 ]
ghci > filter even [ 1 .. 10 ]
[ 2 , 4 , 6 , 8 , 10 ]
ghci > let notNull x = not ( null x ) in filter notNull [[ 1 , 2 , 3 ], [] , [ 3 , 4 , 5 ], [ 2 , 2 ], [] , [] , [] ]
[[ 1 , 2 , 3 ], [ 3 , 4 , 5 ], [ 2 , 2 ]]
ghci > filter ( ` elem ` [ 'a' .. 'z' ]) "or laugh at mE Because I aM different"
"uagameasadifeent"
ghci > filter ( ` elem ` [ 'A' .. 'Z' ]) "I Laugh At You Because ur All the Same"
"GAYBALLS"
```

All of this could also have been accomplished with comprehension lists using predicates. There is no rule that says when to use map or filter instead of comprehension lists, you simply have to decide which is more readable depending on the context. The equivalent filter of applying multiple predicates on a list by comprehension is the same as applying multiple filters or joining predicates using the && logical function .

We use comprehension lists to filter out elements that were less than or equal to and greater than the pivot. We can achieve the same in a more readable way using filter .

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

Mapping and filtering are the daily bread of all the tools of a functional programmer. It doesn't matter if you use the map and filter functions or lists for understanding. Remember how we solved the problem of finding right triangles with a certain circumference. In imperative programming, we should have solved the problem by nesting three loops and then checking if

the current join satisfies the properties of a right triangle. In that case, we would have shown it on screen or something similar. With functional programming this pattern is achieved with mapping and filtering. You create a function that takes a value and produces a result. We map that function over all the items in the list and then filter the resulting list to satisfy our search. Thanks to Haskell's lazy evaluation, even if you map something on a list multiple times or filter it multiple times, it will only cycle through the list once.

Let's find the **largest number below 100,000 that is divisible by 3829** . To achieve this, we simply filter a set of possibilities in which we know the solution is.

```
largestDivisible :: ( Integral a ) => a
largestDivisible = head ( filter p [ 100000 , 99999 .. ])
    Where p x = x ' mod ' 3829 == 0
```

First we create a list of numbers less than 100,000 in descending order. Then we filter it with our predicate and since the numbers are sorted in descending order, the largest number that satisfies our predicate is the first item in the filtered list. We don't even have to use a finite list for our starting set. The lazy evaluation appears again. Since in the end we only end up using the head of the list, it doesn't matter if the list is finite or infinite. The evaluation stops when the first suitable solution is found.

Next, let's find the **sum of all the odd squares that are less than 10,000** . But first, since we are going to use it in our solution, we are going to introduce the takeWhile function . Take a predicate and a list and loop through the list from the beginning and return these elements as long as the predicate stays true. Once you find a predicate that doesn't evaluate to true for. If we get the first word of "The elephants know how to ride one party" , we could do takeWhile (/ = ' ') "The elephants know how to ride one party" and would obtain "The" . Okay, now for the sum of all the odd squares less than 10,000. First we will start mapping the function (^ 2) to the infinite list [1 ..] . Then we filter the list to stay only with the odd ones. Then we take the elements as long as they are less than 10,000. Finally, we obtain the sum of all these elements. We don't even have to create a function to get the result, we can do it in one line in GHCi:

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

166650

Awesome! We start with some initial data (the infinite list of natural numbers) that we map, filter, and then trim until they fit our needs and then add them together. We could also have written this using comprehension lists.

```
ghci > sum ( takeWhile ( < 10,000 ) [ n ^ 2 | n <- [ 1 .. ], odd ( n ^ 2 )])
166650
```

It's a matter of taste. Again, Haskell's characteristic lazy evaluation is what makes this possible. We can map and filter an infinite list since it will not actually map it or filter it to the end, it will delay these actions. Only when we force Haskell to show us the sum does he make the sum of which tells takeWhile that he needs those numbers. takeWhile forces mapping and filtering, but only until it finds a number greater than or equal to 10,000.

In our next problem we are going to deal with the Collatz sequences. We take a natural number. If that number is even we divide it by two. If it is odd, we multiply it by three and add one to it. We take the resulting number and apply the same to it, which produces a new number and so on. Summarizing, we get a sequence of numbers. It is known that for every number the sequence ends with one. So we start with the number 13, we get this sequence: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. 13 * 3 + 1 equals 40. 40 divided by two is 20, etc. . We can see that the sequence has 10 terms. Now, what we want to know is: for each number between 1 and 100, how many sequences have a length greater than 15? First of all we create a function that produces a sequence:

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

Since the sequence ends in 1, that's the base case. It is a typical recursive function.

```
ghci > chain 10
[ 10 , 5 , 16 , 8 , 4 , 2 , 1 ]
ghci > chain 1
[ 1 ]
ghci > chain 30
```

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

Well! It seems to work correctly. And now, the function that gives us the answer to our problem:

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

We map the list [1..100] with the function chain to obtain the list of the sequences. Then we filter the list with a predicate that simply tells us if a list is larger than 15. Once we have filtered, we see how many sequences are left in the resulting list.

Note

This function has the type numLongChains :: Int because length returns the type Int instead of a Num for historical reasons.

We can also do things like map (*) [0 ..] , with the sole purpose of illustrating how currification works and how functions (partially applied) are real values that can be passed as parameters in other functions or as they can be included in lists (only you can't show them on screen). So far we have only mapped over lists functions that take a single parameter, such as map (* 2) [0 ..] to get a list of type (Num a) => [a] , but we can also use map (*) [0 ..] without any problem. What happens is that each number in the list is applied to * that has the type (Num a) => a -> a -> a . Applying a single parameter to a function that has two parameters we get a function that only takes one parameter, so we would have a list of functions (Num a) => [a -> a] . map (*) [0 ..] `` produces a list that we could write as `` [(0 *), (1 *), (2 *), (3 *), (4 *), (5 *) ...

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

By obtaining the 4th element of our list we obtain a function equivalent to (4 *) . And then we apply 5 to that function. So it's actually like we write (4 *) 5 or just 4 * 5 .

Lambdas

Lambdas are anonymous functions that are usually used when we need a function only once. We usually create lambda functions for the sole purpose of passing them to higher order functions. To create a lambda we write a \ (Because it has a certain resemblance to the Greek letter lambda if you give it a lot of imagination) and then the parameters separated by spaces. Then we write a -> and then the body of the function. We usually wrap them in parentheses as they would otherwise extend to the rest of the line.

If you look 10 cm above you will see that we use a where section in our numLongChains function to create the isLong function for the sole purpose of using it in a filter. Well, instead of doing that we can use a lambda:

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

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

It is very common for people who are not very used to how currification and partial application work using lambdas when they shouldn't. For example, the expression map (+3) [1,6,3,2] and map (\ x -> x + 3) [1,6,3,2] are equivalent since both expressions, (+3) y (\ x -> x + 3) are functions that take a number and add 3. Nothing more to say, creating a lambda in this case is stupid since the partial application is much more readable.

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

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

And just like normal functions, lambdas can use pattern matching. The only difference is that you cannot define multiple patterns for one parameter, such as creating [] and (x: xs) for the same parameter so that the variables fit one or the other. If pattern matching fails on a lambda, it will throw a runtime error, so be careful when using them.

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

We usually surround lambdas with parentheses unless we want them to extend to the end of the line. Here's something interesting, because the functions are curred by default, these two definitions are the same:

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

If we define functions in this way, it is obvious why type definitions are the way they are. There are three -> in both the type declaration and the equation. But of course, the first way of writing functions is much more readable, and the second way is only to illustrate currification.

However there are times when it is more interesting to use this notation. I think the flip function is much more readable if we define it like this:

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

Although it is the same as writing flip ' f x y = f y x , we make it obvious that most of the type we will use to produce a new function. The most common use case for flip is to call it with just the parameter function and then pass the resulting function as a parameter to map`` or `` filter . So use lambdas when you want to make it explicit that your function is primarily intended to be partially applied and passed as a function as a parameter.

Folds and origami

Going back to when we were dealing with recursion, we realized that many functions operated with lists. We used to have a base case that was the empty list. We had to use an x: xs pattern and do some operation with a single item in the list. This suggests that it is a very common pattern, so a few very useful functions were created to encapsulate this behavior. These functions are called folds (or *folds* in English). They are a kind of map function , only they reduce the list to a single value.

A fold takes a binary function, an initial value (I like to call it the accumulator), and a list to fold. The binary function takes two parameters by itself. The binary function is called with the accumulator and the first (or last) element and produces a new accumulator. Then the binary function is called back next to the new accumulator and the new first (or last) item in the list, and so on. When the entire list has been scrolled, only one accumulator remains, which is the value to which the list has been reduced.

Let's first look at the foldl function , also called fold from the left. It folds the list starting from the left. The binary function is applied together with the initial value and the head of the list. This produces a new accumulator and the binary function is called again with that new value and the next element, etc.

We're going to redeploy sum , only this time, we're going to use a fold instead of an explicit recursion.

```
sum ' :: ( Num a ) => [ a ] -> a
sum ' xs = foldl ( \ acc x -> acc + x ) 0 xs
Testing one, two, three:
ghci > sum ' [ 3 , 5 , 2 , 1 ]
eleven
```

Let's take a look at how this fold works. \ acc x -> acc + x is the binary function. 0 is the initial value and xs is the list to be folded. First, 0 is used as the parameter acc the binary function and 3 is used as the parameter x (or current value) .` '0 + 3 produces a 3 which becomes the new accumulator. Then 3 is used as the accumulator and 5 as the current element and therefore 8 becomes the new accumulator. We go ahead and 8 is the accumulator, 2 the current item, so the new accumulator is 10 . To finish that 10 is used as accumulator and 1 as the current element, producing a 1 . Congratulations, you have made a fold!

On the left you have a professional diagram that illustrates how a fold works step by step. The green numbers (if you see them yellow you may be colorblind) are the accumulators. You can see how the list is consumed by the accumulator from top to bottom. Yum, yum, yum ... If we consider that the functions are curred, we can write this implementation more beautifully as:

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

The lambda function (\ acc x -> acc + x) is the same as (+) . We can omit the xs parameter since calling foldl (+) 0 returns a function that takes a list. Generally, if you have a function of the type foo a = bar b a you can write it as foo = bar b thanks to curing.

We are going to implement another function with a fold on the left before continuing with the folds on the right. I'm sure you know that elem checks if an item is part of a list so I won't explain it again (mmm ... I think I already have). We are going to implement it.

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

Well, well, well ... What are we doing here? The start value and accumulator are both of the Boolean type. When we talk about folds both the type of the accumulator and the type of the final result are the same. We start with the initial value False . It makes sense since we assume the item is not in the list. Also because if we call a fold with an empty list the result will be simply the initial value. Then we check if the current element is the one we are looking for. If it is, we set the accumulator to True . If it is not, we leave the accumulator as it was. If it was already False , it remains in that state since the current element is not the one we are looking for. If it was True , it stays as it was too.

Folds on the right now work the same as folds on the left, except that the accumulator consumes elements on the right. The binary function of the folds on the left as the first parameter the accumulator and the current value as the second parameter (such that thus: \ acc x -> ...), the binary function of the folds on the right has the current value as first parameter and the accumulator afterwards (like this: \ x acc -> ...). It makes sense since the fold on the right has the accumulator on the right.

The accumulator (and therefore the result) of a fold can be of any type. It can be a number, a boolean, and even a new list. We are going to implement the map function with a fold to the right. The accumulator will be a list, in which we will accumulate the elements of the list already mapped. It is obvious that the initial value will be an empty list.

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

If we are mapping (+3) to [1,2,3] , we scroll through the list from the right side. We take the last element, which is 3 and apply the function to it, so that it ends up being a 6 . Then we add it to the accumulator which is [] . 6: [] is [6] which becomes the new accumulator. We apply (+3) to 2 , that is five and is added (:) to the accumulator, so that remains [5,6] . We do the same with the last element and we end up getting [4,5,6] .

Of course, we could also have implemented this function using a fold from the left. It would be something like map ' f xs = foldl (\ acc x -> acc ++ [f x]) [] xs , but the point is that the function ++ is far less efficient than : , so we usually use folds for right when we build lists from a list.

If you turn a list inside out, you can make a fold on the right as if it were a fold on the left and vice versa. Sometimes you don't even have to. The sum function for example can be implemented with both a fold on the left and on the right. One big difference is that the folds on the right work with infinite lists, while the folds on the left do not. To clarify things, if you take an infinite list somewhere and apply a fold to it to the right, at some point it will reach the beginning of the list. However, if you take an infinite list at some point and apply a fold to the left it will never reach the end.

Folds can be used to implement any function that loops through a list, item by item, and then returns a value. Whenever you want to loop through a list and return a value, there are possibilities to use a fold . This is why folds, along with mappings and filters, are some of the most useful functions in functional programming.

The foldl1 and foldr1 functions are very similar to foldl and foldr , only instead you don't need to enter a start value. They assume that the first (or last) item in the list is a start value, then they start folding the list by the next item. This reminds me that the sum function can be implemented as: sum = foldl1 (+) . Since these functions depend on the lists to be folded have at least one element, they can cause runtime errors if they are called with empty lists. On the other hand, both foldl and foldr work well with empty lists. When you make a fold, think carefully about how to act before an empty list. If the function doesn't make sense when called with empty lists you can probably

use foldl1`` and `` foldr1 to implement it.

For the sole reason of showing you how powerful these functions are, we're going to implement a handful of standard functions using folds:

```
maximum ' :: ( Ord a ) => [ a ] -> a
maximum ' = foldr1 ( \ x acc -> if x > acc then x else acc )

reverse ' :: [ a ] -> [ a ]
reverse ' = foldl ( \ acc x -> x : acc ) []

product ' :: ( Num a ) => [ a ] -> a
product ' = foldr1 ( * )

filter ' :: ( a -> Bool ) -> [ a ] -> [ a ]
filter ' p = foldr ( \ x acc -> if p x then x : acc else acc ) []

head ' :: [ a ] -> a
head ' = foldr1 ( \ x _ -> x )

last ' :: [ a ] -> a
last ' = foldl1 ( \ _ x -> x )
```

head is better implemented with pattern wrapping, but this way you can see that it can even be implemented with folds. Our reverse ' function is pretty clear I think. We take the empty list as the starting value and then we go through the list from the left and simply add elements to our accumulator. At the end we have the list backwards. \ acc x -> x : acc looks like the function : only the parameters are backwards. For this reason we could also have written this: foldl (flip (:)) [] .

There is another way to represent the folds on the left and on the right. Let's say we have a right fold, a function f, and a start value z . If we make the fold over the list [3,4,5,6] , it is basically as if we made f 3 (f 4 (f 5 (f 6 z))) . f is called with the last item in the list and the accumulator, that value is given as the accumulator of the next call and so on. If we take + as f and a start value 0 , we have 3 + (4 + (5 + (6 + 0))) . Represented by prefix would be (+) 3 ((+) 4 ((+) 5 ((+) 6 0))) . Similarly if we make a fold from the left, we take g as a binary function and z as an accumulator, it would be equivalent to making g (g (g (g z 3) 4) 5) 6 . If we take flip (:) as a binary function and [] as the accumulator (so we're reversing the list), then it would be equivalent to flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6 . And I am almost sure that if you evaluate this expression you will get [6,5,4,3] .

scanl and scanr are like foldl and foldr , only they return all intermediate accumulators as a list. There are also scanl1 and scanr1 , which are similar to foldl1 and foldr1 .

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

When we use scanl , the final result will be the last element of the resulting list, while with scanr it will be at the beginning.

These functions are used to monitor the progression of a function that can be implemented with a fold. Let's answer the following question. How many elements does the sum of all the roots of all the natural numbers take to exceed 1000? To obtain the roots of all natural numbers we simply make map sqrt [1 ..] . Now to get the sum you could use a fold but since we are interested in the progression of the sum we will use scanl . When we get the resulting list, we simply count how many sums are below 1000. The first sum in the list will be 1. The second will be 1 plus the root of 2. The third will be the same as the previous one plus the root of 3. If there are X sums less than 1000, so it will take X + 1 elements for the sum to exceed 1000.

```
sqrtSums :: Int
sqrtSums = length ( takeWhile ( < 1000 ) ( scanl1 ( + ) ( map sqrt [ 1 .. ]))) + 1
ghci > sqrtSums
131
ghci > sum ( map sqrt [ 1 .. 131 ])
1005.0942035344083
ghci > sum ( map sqrt [ 1 .. 130 ])
993.6486803921487
```

We use takeWhile instead of filter because it doesn't work with infinite lists. Even though we know that the list is ascending, filter doesn't know it, so we use takeWhile to cut the list for the first occurrence of a sum that exceeds 1000.

Application of functions with $

Okay, now let's look at the $ function , also called a function application. First of all let's see how it is defined:

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

But what...? Why do we want such a useless operator? It is simply the application of a function! Well, almost, but not only that. While the normal function application (a space between two things) has a high order of precedence, the $ function has the lowest order of precedence. The application of functions with space is associative to the left (so f a b c is the same as ((f a) b) c), the application of functions with $ is associative to the right.

That is all very well, but what use is this to us? Basically it's a convenience function that we use so we don't have to type a lot of parentheses. Consider the expression sum (map sqrt [1..130]) . Thanks to $ having a low order of precedence we can write the same expression as sum $ map sqrt [1..130] , saving our fingers from pressing those annoying keys. When a $ is found , the expression on the right is applied as a parameter to the function on the left. What about sqrt 3 + 4 + 9 ? This expression adds 4 plus 9 plus the root of 3. If what we want is the root of 3 + 4 + 9 we have to write sqrt (3 + 4 + 9) or if we use $ we can write it as sqrt $ 3 + 4 + 9 since $ has a lower order of precedence than any other operator. For this reason we can imagine $ as a kind of open parenthesis that automatically adds a closing to the end of the expression.

What would happen to sum (filter (> 10) (map (* 2) [2..10])) ? Well, since $ is right associative, f (g (z x)) would be the same as f $ g $ z x . We go ahead and sum (filter (> 10) (map (* 2) [2..10])) can be written as sum $ filter (> 10) $ map (* 2) [2..10] .

But apart from removing the parentheses, the existence of the $ operator also means that we can treat the application of functions as one more function. In this way, we can, for example, map a list of functions:

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

Composition of functions

In mathematics the composition of functions is defined as:, which means that composing two functions creates a new one that, when called with a parameter, say *x* , is equivalent to calling *g* with *x* and then calling *f* with the result previous.

In Haskell the composition of functions is practically the same. We perform the composition of functions with the function . , which is defined as:

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

Look at the type declaration. f must have as a parameter a value with the same type as the value returned by g . So the resulting function takes a parameter of the same type that g takes and returns a value of the same type that f returns . The expression negate . (-3) returns a function that takes a number, multiplies it by three, and then negates it.

One of the uses of function composition is to create functions on the fly to be passed on to other functions. Sure, you can use lambdas but many times the composition of functions is clearer and more concise. Let's say we have a list of numbers and we want to make them all negative. One way to do this would be to first get the absolute number and then deny it, something like this:

```
ghci> map (\ x -> negate (abs x)) [5, -3, -6.7, -3.2, -19.24]
[-5, -3, -6, -7, -3, -2, -19, -24]
```

Notice that the lambda function resembles the definition of composition of functions. Using the composition of functions it would look like this:

```
ghci> map (negate. abs) [5, -3, -6.7, -3.2, -19.24]
[-5, -3, -6, -7, -3, -2, -19, -24]
```

Great! The composition of functions is associative to the right, so we can compose several functions at the same time. The term f (g (z x)) is equivalent

to (f . G . Z) x . With this in mind, we can convert:

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

In this:

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

And what about functions that take various parameters? Well, if we want to use them in the composition of functions, we have to partially apply them so that each function takes a single parameter. sum (replicate 5` '(max 6.7 8.9)) can be written as (sum . replicate 5 . max 6.7) 8.9 or sum . replicate 5 . max 6.7 $ 8.9 . What happens here is: a function is created that takes max 6.7 and applies replicate 5 to it. Then another function is created that takes the result of the above and performs a sum. Finally, the above function is called with 8.9 . Normally it reads as: Apply 8.9 to max 6.7 , then apply replicate 5 and then apply sum to the previous result. If you want to rewrite an expression with a lot of parentheses using function composition, you can start by putting the last parameter of the outermost function after $, and then start composing all the other functions, writing them without the last parameter, and putting . between them. If you have replicate 100 (product (map (* 3) (zipWith max [1,2,3,4,5] ` ` [4,5,6,7,8]))) you can also write it as replicate 100 . product . map (* 3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8] . If an expression ends with 3 parentheses, there are possibilities to write the same expression using 3 function compositions.

Another common use of function composition is to define functions in the so-called dot-free style. Take a look at this function we wrote earlier:

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

Note

The term *freestyle points* (*point-free style* or *pointless style* English) originated in topology , a branch of mathematics that works with spaces

compounds of points and functions between these spaces. So a point-free style function is a function that doesn't explicitly mention the points (values) of the space it acts on. This term can be confusing to people since normally the freestyle of points implies to use the operator of composition of functions, which is represented with a point in Haskell.

xs is exposed on both sides of the equation. We can remove xs from both sides thanks to curing , since foldl (+) 0 is a function that takes a list. Writing the above function as sum ' = foldl (+) 0 is called a dot-free style. How do we write this in dot free style?

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

We cannot simply remove x from both sides. The x in the body of the function has a parenthesis after it. cos (max 50) doesn't make much sense. You cannot calculate the cosine of a function. What we do is express fn as a composition of functions.

```
fn = ceiling . refuse . so . cos . max 50
```

Excellent! Many times a composition of functions is much more concise and readable, since it makes you think about functions and how the parameters are passed between them instead of thinking about the data and how they are transformed. You can use simple functions with function composition to create much more complex functions. However, many times, writing a function in dot-free style can be less readable if the function is very complex. That is why the use of function composition for very long function chains is discouraged. The recommended style for these cases is to use let sections to give names to intermediate results, dividing the problem into sub-problems and then composing with all of them so that if someone reads it, it will make sense.

In the mappings and filters section, we solved the problem of finding the sum of all odd squares less than 10,000. Here is what the solution would look like if we put it in a function:

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

Being a fan of feature composition, I could probably have written it as:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile ( < 10,000 ) . filter odd . map ( ^ 2 ) $ [ 1 .. ]
```

However, if there is a chance that someone else would read this code, you could write it as:

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

It wouldn't win any short code competition, but it would make life easier for someone who had to read it.

Chapter V

Modules

Loading modules

A Haskell module is a collection of related functions, types, and type classes. A Haskell program is a collection of modules where the main module loads other modules and uses the functions defined in them to do something. Having the code divided into modules has many advantages. If a module is generic enough, the functions that are exported can be used in a wide variety of programs. If your code is separated into modules that don't depend much

on each other (we also say that they are loosely coupled), then you can reuse them. This makes the task of programming more manageable since it is all divided into several parts, each with its own purpose.

Haskell's standard library is divided into several modules, each one consisting of functions and types that are related in some way and serve a common purpose. There is a module for manipulating lists, a module for concurrent programming, a module for working with complex numbers, etc. All the functions, types and type classes we have worked with so far are part of the `Prelude` module , which is imported by default. In this chapter we are going to see a few useful modules and their respective functions. But first, let's see how the modules are imported.

The syntax for importing modules into a Haskell script is `import <module name>` . It must appear before we define any function, so module imports are usually at the beginning of files. A script can obviously import multiple modules. Simply put each `import` on separate lines. We're going to import the `Data.List` module , which contains a handful of useful functions for working with lists, and we'll use a function that exports that module to create a function that tells us how many unique elements are in a list.

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length. Nub
```

When we `import Data.List` , all the functions that `Data.List` exports are available in the global namespace. This means that we can access all these functions from our script. `nub` is a function that is defined in `Data.List` which takes one list and returns another without duplicate elements. Composing `length` and `nub` by making `length. nub` produces a function equivalent to `\ xs -> length (nub xs)` .

You can also import modules and use them when we are working with GHCi. If we are in a GHCi session and we want to use the functions that `Data.List` exports we do this:

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

If we want to load several modules within GHCi we don't have to use `: m +` several times, we can simply load several modules at once:

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

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

If you only need some functions of a module, you can select them so that only those functions are imported. If we want to import only the `nub` and `sort` functions of `Data.List` we do the following:

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

You can also import all the functions of a module except some selected ones. Normally this is used when we have several modules that export a function with the same name and we want to get rid of one of them. Let's say we already have a function called `nub` and we want to import all the `Data.List` functions except the `nub` function :

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

Another way to deal with name collisions is with qualified imports. The `Data.Map` module , which offers a data structure to search for values by key, exports a bunch of functions with names equal to the `Prelude` functions , such as `filter` or `null` . So when we import `Data.Map` and call `filter` , Haskell doesn't know which function to call. Here is how we solve it:

```
import qualified Data.Map
```

In this way, if we want to refer to the `filter` function of `Data.Map` , we have to use `Data.Map.filter` , while if we simply use `filter` we are referring to the normal filter that we all know. Writing `Data.Map in` front of all functions is quite cumbersome. For this reason we can rename a qualified import with something a little shorter:

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

Thus, to refer to the `filter` function of `Data.Map` we only have to use `M.filter` .

You can use this handy reference to see what modules are in the standard library. One way to get information about Haskell is to simply click on the standard library reference and explore its modules and functions. You can also see the source code of each module. Reading the source code of some modules is a very good way to learn Haskell.

You can search for features or find where they are located using Hoogle . It is an amazing Haskell search engine. You can search by function name, module name or even by the type definition of a function.

Data.List

The `Data.List` module deals exclusively with lists, obviously. It offers very useful functions for working with lists. We have already used some of these functions (like `map` and `filter`) since the `Prelude` module exports some functions from `Data.List` for convenience. It is not necessary to import the `Data.List` module in a qualified way because it does not collide with any `Prelude` name except for those already taken by the `Data.List` . Let's take a look at some functions that we have not yet met.

- `intersperse` takes one element and a list puts that element between each pair of elements in the list. A demonstration:

```
ghci> intersperse '.' "MONKEY"
"MONKEY"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

- `collate` takes a list and a list of lists. Insert the first list among all the other lists, resulting in a single list.

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

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

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

Suppose we have the polynomials , and and want to add them. We can use the lists `[0,3,5,9]` , `[10,0,0,9]` and `[8,5,1, -1]` to represent them in Haskell. Now, to add them all we have to do is:

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

[18,8,6,17]

When we transpose these three lists, the cubic powers are in the first row, the squares in the second row, etc. Mapping `sum` over this produces the result we are looking for.

-

`foldl'` and `foldl1'` are strict versions of their respective lazy versions. When we use lazy folds on very large lists we often get stack overflow errors. The reason this happens is that due to the nature of lazy folds, the accumulator value is not updated until the fold is made. What actually happens is that the accumulator makes a kind of promise that he will calculate the value when asked to produce a result (this is also called thunk). This happens for each intermediate value of the accumulator and all those *thunks* overflow the stack. Strict folds do not suffer from this error as they actually compute intermediate values as they run through the list instead of filling the stack with *thunks* . You know, if you ever run into stack overflow errors while folding, try these features.

- `concat` flattens a list of lists into a simple list with the same elements.

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

It basically removes a level of nesting. If you want to fully flatten `[[[2,3], [3,4,5], [2]], [[2,3], [3,4]]]` , which is a list of lists of lists, you have than flatten it twice.

- `concatMap` is the same as first mapping a function to a list and concatenating all the results.

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

- `and` takes a list of booleans and returns `True` only if all the elements in the list are `True` .

- ghci> and $ map (> 4) [5,6,7,8]

- True
- ghci> and $ map (== 4) [4,4,4,3,4]
- False

- or is like and only that returns True only if there is any True element in the list.

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

- any and all take a predicate and a list and check if the predicate is satisfied for some or all of the elements respectively. Normally we use these functions instead of having to map a list and then use and or or .

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

- iterate takes a function and an initial value. Apply that function to the initial value, then apply the function to the previous result, then apply the same function to the previous result again, etc. Returns all results in infinite list form.

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

- splitAt takes a number and a list. Then it divides the list by the indicated index and returns a pair with both lists.

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

- takeWhile is a really useful function. It takes element from a list as long as the predicate is held true, and then when it finds an element that doesn't satisfy the predicate, it cuts the list.

- ghci> takeWhile (> 3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
- [6.5.4]
- ghci> takeWhile (/ = '') "This is a sentence"
- "Este"

Let's say we want to know the sum of all cubic powers that are below 10,000. We cannot map `(^ 3)` to `[1 ..]` , apply a filter and then add the result since filtering an infinite list never ends. You know that all elements are ascending but Haskell does not. So we use this:

```
ghci> sum $ takeWhile (<10,000) $ map (^ 3) [1 ..]
53361
```

We apply `(^ 3)` to an infinite list and once an element greater than 10,000 is found, the list is cut off. That way, we can then easily add up the list.

- `dropWhile` is similar, only it discards element while the predicate is fulfilled. Once the predicate is evaluated to `False` , it returns the rest of the list. A lovely feature!

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

They give us a list that represents the values of the shares by dates. The list contains 4-tuples whose first element is the value of the action, the second the year, the third the month and the fourth the day. If we wanted to know when an action first reached $ 1,000, we could use this:

```
ghci> let stock = [(994.4,2008,9,1), (995.2,2008,9,2), (999.2,2008,9,3),
(1001.4,2008,9,4), (998.3,2008 , 9.5)]
ghci> head (dropWhile (\ (val, y, m, d) -> val <1000) stock)
(1001.4,2008,9.4)
```

- `span` is a type of `takeWhile` , only it returns a pair of lists. The first list contains everything the resulting `takeWhile` list would have . The list would contain the entire list that had been cut.

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

- While `span` splits the list when the predicate is no longer true, `break` splits the list when the predicate is first fulfilled.

Equivalent to `span (not. P)` .

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

When we use `break` , the second list will start with the first element that satisfies the predicate.

- `sort` simply sorts a list. The type of elements contained in the list must be a member of the `Ord` type class , since if the elements of the list cannot be put in some sort of order, the list cannot be sorted.

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

- `group` takes a list and groups adjacent elements that are the same into sublists.

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

If we order a list before grouping it, we can obtain how many times each element is repeated.

```
ghci> map (\ l @ (x: xs) -> (x, length l)). group. sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4), (2,7), (3,2), (5.1), (6.1), (7.1)]
```

- `inits` and `tails` are like `init` and `tail` , only they are applied recursively until nothing is left in the list. Observe:

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

We are going to use a fold to implement a search for a sublist within a list.

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

First we call `tails` with the list we are looking for. Then we go

through each queue and see if it starts with what we are looking for.

- With this, we have actually created a function that behaves like `isInfixOf` . `isInfixOf` searches for a sublist in a list and returns `True` if the sublist we are looking for is somewhere in the list.

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

- `isPrefixOf` and `isSuffixOf` search for a sublist from the beginning or the end of a list, respectively.

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

- `elem` and `notElem` check if an element is inside a list.
- `partition` takes a list and a predicate and returns a pair of lists. The first list contains all the elements that satisfy the predicate, the second all those that do not.

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

It is important to know the differences that this function has with `span` and `break` .

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

Both `span` and `break` end when they find the first element that satisfies or does not satisfy the predicate, `partition` goes through the entire list and divides it according to the predicate.

- `find` takes a list and a predicate and returns the first element that satisfies the predicate. But, it returns the wrapped element in a `Maybe` value . We will take a closer look at algebraic data

types in the next chapter, but for now this is all you need to know: a `Maybe` value can be either a `Just something` or `Nothing`. In the same way that a list can be either an empty list or one with elements, a `Maybe` value can be either an element or none. And since the type of the list says that, for example, a list of integers is `[Int]`, the type of a `Maybe` that contains an integer is `Maybe Int`. Anyway, let's see the `find` function in action.

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

Look at the `find` type. Its result is of type `Maybe a`. This is similar to having something of type `[a]`, except that a value of type `Maybe` can only have either one element or none, while a list can have no element, a single element, or several of them.

Remember when we were looking for the first time that a stock was over $ 1,000. We use `head (dropWhile (\ (val, y, m, d) -> val <1000) `stock`. Also remember that `head` is not a safe function. What if we had never reached $ 1000? `DropWhile` would have returned a list empty and applying `head` to an empty list only gives one result, an error, however if we use `find (\ (val, y, m, d) -> val> 1000) stock`, we can be much calmer. they never exceed 1000 $ (that is, no element satisfies the predicate), we will get `Nothing`, and if they do we will get a valid answer, like `Just (1001.4,2008,9,4)`.

- `elemIndex` is similar to `elem`, only it does not return a boolean value. Maybe it returns the index of the item we are looking for. If item is not in the list it will return `Nothing`.

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

- `elemIndices` is like `elemIndex`, only it returns a list of indexes in case the element we are looking for appears several times through the list. Since we are using a list to represent the

indexes, we don't need the `Maybe` type , since the case that nothing is found can be represented with the empty list, which is synonymous with `Nothing` .

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

- `findIndex` is like `find` , only it can return the index of the first element that satisfies the predicate. `findIndices` returns the index of all the elements that satisfy the predicate in list form.

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

- We have already talked about `zip` and `zipWith` . We saw that these functions combined two lists, either with a pair or with a binary function (in the sense that it takes two parameters). What if we want to combine three lists? Or combine three lists with a function that takes three parameters? Well, for that we have `zip3` , `zip4` , etc. and `zipWith3` , `zipWith4` , etc. These variants go up to 7. This may seem like some kind of arrangement, but it works very well in reality, since there are not so many occasions when we want to combine 8 lists. There is also a very clever way to combine an infinite number of lists, but we have not yet advanced enough to explain it here.

- ghci> zipWith3 (\ xyz -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
- [7,9.8]
- ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
- [(2,2,5,2), (3,2,5,2), (3,2,3,2)]

Like the other functions, the resulting lists are as long as the shortest list.

- `lines` is a very useful function when dealing with some type of input, such as files. Take a string and return each line of that separate string in a list.

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

`'\ n'` is the character that represents the unix line break. The backslashes have a special meaning in the Haskell strings and

characters.

- unlines is the inverse function of lines . Take a list of strings and join them using a '\ n' .

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

- words and unwords are used to separate separate a line of text by words. Very useful.

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

- We have seen nub before . It takes a list and removes the repeating elements, returning a list in which each element is unique. This function has a very strange name. Turns out, nub means a small part or an essential part of something. In my opinion, I think they should use real names for functions instead of ancestral words.

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

- delete takes an item and a list and removes the first identical item from that list.

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

- \\ is the division function. It works like a division basically. Remove the first occurrence from the list on the right of the items in the list on the left.

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

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

- union works like the union of sets. Returns the union of two

lists. Basically it loops through each item in the second list and adds it to the first list if it's not already contained. Be careful, duplicates are only removed from the first list.

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

- `intersect` works like the intersection of sets. Returns the elements that are in both lists.

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

- `insert` takes an element and a list that can be sorted and inserts this element in the last position where it is less than or equal to the next element. In other words, `insert will loop` through the list until it finds an element larger than the element we are inserting, and will insert it before that element.

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

The `4` is inserted just after the `3` and before the `5` in the first example, and between `3` and `4` in the second.

If we use `insert` to insert something into an ordered list, the result will continue to be ordered.

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

What `length` , `take` , `drop` , `splitAt` , `!!` and `replicate they` have in common is that they take an `Int` as a parameter (or return it), even though these functions could be more generic and useful if they simply took any type that was part of the `Integra` or `Num` type classes (depending on the functions) . They do it for historical reasons. Probably if they fixed this much existing code would stop working. This is why `Data.List` has its own more generic variants, they are called `genericLength` , `genericTake` , `genericDrop` , `genericSplitAt` , `genericIndex` and `genericReplicate` . For example, `length` has the type `length :: [a] -> Int` . If we try to

get the mean of a number list using `let xs = [1..6] in sum xs / length xs` we will get a type error since we cannot use `/` with an `Int` . On the other hand `genericLength` has the type `genericLength :: (Num a) => [b] -> a` . Since `Num` can behave like a floating point number, getting the mean by doing `let xs = [1..6] in sum xs / genericLength xs` works perfectly.

The `nub` , `delete` , `union` , `intersect` and `group` functions have their respective general functions called `nubBy` , `deleteBy` , `unionBy` , `intersectBy` and `groupBy` . The difference between them is that the first set of functions uses `==` to check equality, while the other set takes an equality function and compares elements using this function. `group` is the same as `groupBy (==)` .

For example, let's say we have a list containing the value of a function for every second. We want to segment the list into sublists based on when a value was below zero and when it was above. If we used a normal `group it` would simply group the adjacent equal values. But what we want is to group them according to whether they are positive or not. This is where enters juengo `groupBy` . The equality function taken by functions with the suffix `By` must take two parameters of the same type and return `True` if they consider that they are equal by their own criteria.

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

In this way we can clearly see which sections are positive and which are negative. The equality function we used only returns `True` when both values are positive or both are negative. This equality function can also be written as `\ xy -> (x> 0) && (y> 0) || (x <= 0) && (y <= 0)` although for my taste the first one is more readable. There is an even clearer way to write equality functions for these functions if we import the `on` function from the `Data.Function` module . `on` is defined as:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \ xy -> f (gx) (gy)
```

So `(==) `on` (> 0)` returns an equality function that behaves the same as `\ xy -> (x> 0) == (y> 0)` . `on` is used a lot with all these functions, since with it, we can

do things like:

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

Very readable! You can read it all at once: Group this by equality in if the elements are greater than zero.

Similarly, the `sort`, `insert`, `maximum`, and `minimum` functions also have their more general equivalents. Functions like `groupBy` take functions that determine whether two elements are equal or not. `sortBy`, `insertBy`, `maximumBy`, and `minimumBy` take a function that determines whether one item is greater than, equal to, or less than another. The `sortBy` type is `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. If you remember, the `Ordering` type can take the `GT`, `EQ` and `LT` values. `sort` is equivalent to `sort compare`, since `comapare` simply takes two elements whose types are in the `Ord` type class and returns their order relationship.

The lists can be compared by lexicographic order. What if we have a list of lists and we don't want to order them based on the content of the interior lists but on their sizes? Well, as you've probably imagined, that's what the `sortBy` function is `for`:

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

Amazing! `compare `on` length`, that reads almost like real English. If you are not sure how `compare `on` length` works here, equivalent to `\ xy -> length x` compare` length y`. When we deal with functions that have the suffix `By` that take equality functions we usually use `(==) `on` something` and when we deal with those that take order functions we usually use `compare` on` something`.

Data.Char

The `Data.Char` module contains what its name suggests. Exports functions that deal with characters. It is also useful when we map or filter strings since they are lists of characters after all.

`Data.Char` exports a handful of character predicates. That is, functions that take

a character and tell us whether an assumption about it is true or false. Here you have them:

- `isControl` checks whether a character is control or not.
- `isSpace` checks if a character is one of the white space characters. That includes spaces, tabs, line breaks, etc.
- `isLower` checks if a character is lowercase.
- `isUpper` checks if a character is uppercase.
- `isAlpha` checks if a character is a letter.
- `isAlphaNum` checks if a character is a letter or a number.
- `isPrim` checks if a character is printable. Control characters, for example, are not.
- `isDigit` checks if a character is a digit.
- `isOctDigit` checks if a character is an octal digit.
- `isHexDigit` checks if a character is a hexadecimal digit.
- `isLetter` checks if a character is a letter.
- `isMark` checks if a character is a Unicode mark. These characters are combined with their adjacent ones.
- `isNumber` checks if a numeric character.
- `isPunctuation` checks if a character is a punctuation mark.
- `isSymbol` checks whether a character is a mathematical symbol or a currency symbol.
- `isSeparator` checks if a character is a space or a Unicode separator.
- `isAscii` checks if a character is one of the first 128 characters in the Unicode character set.
- `isLatin1` checks if a character is one of the first 256 characters in the Unicode character set.
- `isAsciiUpper` checks if a character is capitalized and is also ascii.
- `isAsciiLower` checks if a character is lowercase and is also ascii.

All these functions have the type `Char -> Bool` . Most of the time you will use them to filter strings or something similar. For example, let's say we are

going to make a program that takes a username and that name can only be made up of alphanumeric characters. We can use the `all` function of the `Data.List` module to determine if the name is correct:

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

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

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

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

Mmm ... well, it does the same as `words` but we leave some elements that contain a single space. What can we do? I know, we are going to filter them.

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

`Data.Char` also exports a data type similar to `Ordering` . The `Ordering` type can have an `LT` , `EQ`, or `GT` value . It is a kind of enumeration. Describe a series of possible results given by comparing two elements. The `GeneralCategory` type is also an enumeration. Represents a series of categories to which a character can belong. The main function to get the category of a character is `generalCategory` . It has the type `generalCategory :: Char -> GeneralCategory` . There are 31 different categories so we are not going to show them, but we are going to play a little with this function.

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
```

ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory "\ t \ nA9? |"
[Space, Control, Control, UppercaseLetter, DecimalNumber, OtherPunctuation, MathSymbol]

Since `GeneralCategory` is part of the `Eq` class, we can do things like `generalCategory c == Space` .

To finish, here are a few functions that convert characters:

- `toUpper` converts a character to uppercase. The spaces, numbers and everything else remain the same.
- `toLower` converts a character to lowercase.
- `toTitle` is similar to `toUpper` except for a few letters.
- `digitToInt` converts a character to an `Int` . For it to work, the character must be between the ranges `'0' .. '9'` , `'a' .. 'f'` and `'A' .. 'F'` .

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

- `intToDigit` is the inverse function of `digitToInt` . Take an `Int` that is in the range `0..15` and convert it to a lowercase character.

```
ghci> intToDigit 15
'F'
ghci> intToDigit 5
'5'
```

- The `ord` and `chr` function converts characters to their respective numerical representations and vice versa.

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

The difference between two two-character `ord` values is equal to the difference between the two in the Unicode table.

Caesar encryption is a primitive method of encrypting messages by shifting each character a fixed number of places in the alphabet. We can create a kind of Caesar cipher ourselves, only

we are not going to stick solely to the alphabet.

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

First, we convert the string to a number list. Then we add a constant amount to each number and convert the list of resulting numbers into another text string. If the composition suits you better, you could have done the same with `map (chr. (+ Shift). Ord) msg` . Let's try to encode some messages.

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

It seems to be well encrypted. Decrypting a message is basically putting the offset characters back in place.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
ghci> encode 3 "Im a little teapot"
"Lp # d # olwwoh # whdsrw"
ghci> decode 3 "Lp # d # olwwoh # whdsrw"
"Im a little teapot"
ghci> decode 5. encode 5 $ "This is a sentence"
"This is a sentence"
```

Data.Map

Association lists (also called dictionaries) are lists that are used to store key-value pairs where order does not matter. For example, we can have an association list to store phone numbers, where the phone numbers would be the values and the names of the people would be the keys. We do not care about the order in which they are stored, we just want to obtain the appropriate number for each person.

The most obvious way to represent an association list in Haskell would be to use a list of pairs. The first component of the pairs would be the key, and the

second the value. Here's an example of a phone number association list:

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

Despite this strange alignment, it is simply a list of string pairs. The most common task when working with association lists is to search for a value by key. We are going to create a function that looks for a value given a key.

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

Very simple. This function takes a key and a list, filters the list so that there are only keys that match the key that was passed to it, gets the first element of the resulting list, and then returns the associated value. But what if the key we are looking for is not in the list? Hmm ... If the key is not in the list, we end up trying to apply `head` to an empty list, so we will have a runtime error. However, we must avoid that our programs break so easily, so we are going to use the `Maybe` type . If we can't find the key, we return `Nothing`, and if we find it, we return `Just something` , where something is the value associated with that key.

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

Look at the type declaration. It takes a key that can be compared for equality, an association list and can return a value. Sounds good.

This is a recursive book function that operates on lists. Base case, split a list into head and tail, recursive call ... This is a classic fold, so let's implement it with a fold.

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

Note

Normally it is better to use a pligue in these cases of standard recursion on lists instead of an explicit recursion since it is more readable and easier to identify. Everyone knows that a fold is taking place when they see a call to foldr , but it takes a little longer to read an explicit recursion.

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

It works perfectly! If we have the number of a girl we obtain said Just number , otherwise we obtain Nothing .

We have just implemented the lookup function of the Data.List module . If we want to obtain the value corresponding to a key, we only have to scroll through the list until we find it. The Data.Tree module offers much more efficient association lists (since they are implemented with trees) and also offers a lot of useful functions. From now on we will say that we are working with dictionaries instead of association lists.

Because Data.Map exports functions that collide with those of Prelude and Data.List , we will import it in a qualified way.

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

Put this statement in a script and then load it with GHCi.

Let's go ahead and see what Data.Map has for us. Here's a basic overview of the features.

- The fromList function takes an association list (in list form) and returns a dictionary with the same associations.
- ghci> Map.fromList [("betty", "555-2938"), ("bonnie", "452-2928"), ("lucille", "205-2928")]
- fromList [("betty", "555-2938"), ("bonnie", "452-2928"), ("lucille", "205-2928")]
- ghci> Map.fromList [(1,2), (3,4), (3,2), (5.5)]
- fromList [(1,2), (3,2), (5.5)]

In case there are duplicate keys in the association list, duplicates are discarded. This is the fromList type declaration :

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

It says it takes a list of pairs k and v and returns a dictionary that associates the keys k with the values v . Note that when we created association lists with normal lists, the keys only had to be equal (their type belonged to the Eq type class) but now they have to be sortable. This is basically a restriction of the Data.Map module . You need the keys to be sortable so that you can organize them in a tree.

You should always use the Data.Map module for key-value associations unless the keys are of the Ord type class .

- empty represents an empty dictionary. It doesn't take any parameters, it just returns an empty dictionary.
- ghci> Map.empty
- fromList []
- insert takes a key, a value and a dictionary and returns a new dictionary exactly the same as the previous one, except that it also contains the new key-value.
- ghci> Map.empty
- fromList []
- ghci> Map.insert 3 100 Map.empty
- fromList [(3,100)]
- ghci> Map.insert 5 600 (Map.insert 4 200 (Map.insert 3 100 Map.empty))
- fromList [(3,100), (4,200), (5,600)]
- ghci> Map.insert 5 600. Map.insert 4 200. Map.insert 3 100 $ Map.empty
- fromList [(3,100), (4,200), (5,600)]

We can implement our own fromList function using only an empty dictionary, insert and a fold. Look:

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

It is a fairly simple fold. We start with an empty dictionary and then we fold the list from the right, inserting new key-value pairs in the accumulator.

- null checks if a dictionary is empty.
- ghci> Map.null Map.empty

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

- `size` gives us the size of a dictionary.

- ghci> Map.size Map.empty
- 0
- ghci> Map.size $ Map.fromList [(2.4), (3.3), (4.2), (5.4), (6.4)]
- 5

- `singleton` takes a key and a value and returns a dictionary that only contains that key.

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

- `lookup` functions as the function `lookup` of `Data.List`, only operates dictionaries instead of lists. Returns `Just something` if it finds the key or `Nothing` otherwise.

- `member` is a predicate that takes a key and a dictionary and tells us if that key is contained in the dictionary.

- ghci> Map.member 3 $ Map.fromList [(3.6), (4.3), (6.9)]
- True
- ghci> Map.member 3 $ Map.fromList [(2,5), (4,5)]
- False

- `map` and `filter` work similarly to their list equivalents.

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

- `toList` is the inverse of `fromList`.

  ghci> Map.toList. Map.insert 9 2 $ Map.singleton 4 3
  [(4.3), (9.2)]

- `keys` and `elems` return a list with all keys or all values respectively. `keys` is equivalent to `map fst. Map.toList` and `elems` is equivalent to `map snd. Map.toList`.

- `fromListWith` is a very interesting function. It acts as `fromList`, only it does not discard any predicate, instead, it uses a function that we pass to it to decide which of them should be added. Let's say a girl can have multiple phone numbers and we have an association list like this:

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

This way if we use `fromList` we will lose some numbers. So we can do this:

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\ number1 number2 -> number1 ++ "," ++ number2) xs
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

In case a duplicate key is found, the function we pass will take care of combining the values of the key. We could also do all the values in the association list unit lists first and then use `++` to combine the numbers.

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\ (k, v) -> (k, [v])) xs
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162", "943-2929", "493-2928"]
```

Very simple. Another case would be if we are creating a dictionary from an association list that contains numbers and that when a duplicate key is found, we want the largest value to be the one that is maintained.

```
ghci> Map.fromListWith max [(2,3), (2,5), (2,100), (3,29), (3,22), (3,11), (4,22), (4, fifteen)]
fromList [(2,100), (3,29), (4,22)]
```

Or we could also have chosen to have these values added:

```
ghci> Map.fromListWith (+) [(2,3), (2,5), (2,100), (3,29), (3,22), (3,11), (4,22), ( 4.15)]
fromList [(2,108), (3.62), (4.37)]
```

- `insertWith` is an `insert` in the same way as `fromListWith` is for

insert fromList . Insert a key-value into a dictionary, but if the dictionary already contains that key, use the function we passed to it to determine what to do.

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

These are just some of the functions that Data.Map contains

Data.Set

The Data.Set module offers us operations with sets. Sets like sets in math. The sets are a type of mixture between the list and data dictionaries. All the elements of a set are unique. And since internally they are implemented with trees (like Data.Map dictionaries) they are ordered. Check if there is an element, insert it, delete it, etc. it is much more efficient than doing it with lists. The most common operations when working with sets are inserting elements, checking if an element exists in the set and converting a set to a list.

As the names exported by Data.Set collide with those of Prelude and Data.List we import it in a qualified way.

Put this statement in a script:

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

And then load the script with GHCi.

Let's say we have two pieces of text. We want to know what characters are used in both pieces.

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

The fromList function works as expected. It takes a list and turns it into a set.

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

fromList "! Tabcdefghilmnorstuvwy"

As you can see the elements are ordered and each element is unique. Now we are going to use the intersection function to see what elements are in both sets.

ghci> Set.intersection set1 set2
fromList "adefhilmnorstuy"

We can use the difference function to see which elements of the first set are not in the second set and vice versa.

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

Or we can see all letters that were used in both texts using union .

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

The null , size , member , empty , singleton , insert and delete functions work as expected.

ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3.5]

It can also be consulted by subsets or own sets. Set A is a subset of B, if B contains all the elements of A. Set A is a proper set of B if B contains all the elements that A contains and none more.

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

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

We can also use the `map` and `filter` functions with them.

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

Sets are normally used to remove duplicate elements from a list so we first make a set with `` `fromList` and then convert it back to a list with `toList` . The function `nub` of `Data.List` already performs this task, but if you 're eliminating duplicate a large list is much more efficient if inserting the elements into a whole and then turn it into a list rather than using `nub` . But `nub` only requires that the elements in the list be of the `Eq` type class , while the elements of the sets must be of the `Ord` class .

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

`setNub` is much faster than `nub` for large lists, but as you can see, `nub` preserves the order in which items appear in the list while `setNub` does not.

Creating our own modules

So far we have seen quite a few interesting modules, but how do we create our own modules? Almost every programming language allows you to divide your code into several files and Haskell is no different. When creating programs, it is good practice that the functions and types that are somehow related are in the same module. In this way, we can easily reuse those functions in other programs by importing those modules.

Let's see how we can create our own module by making a small module that exports functions that allow us to calculate the volume and area of a few

geometric objects. We will start by creating a file called Geometry.hs .

We say that a module exports functions. Which means that when we use a module, we can see the functions that this module exports. You can define functions that are called internally, but we can only see the functions that you export.

We specify the name of a module at the beginning of the module. If we have called the file Geometry.hs we must give the name of Geomtry to our module. Then we specify the functions that are exported, and then we start defining those functions. So we start with this.

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

As we observe, we are going to calculate the area and volume of the spheres, cubes and hexahedrons. Let's go ahead and define these functions:

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

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume abc = rectangleArea ab * c

cuboidArea :: Float -> Float -> Float -> Float
```

```
cuboidArea abc = rectangleArea ab * 2 + rectangleArea ac * 2 + rectangleArea cb * 2

rectangleArea :: Float -> Float -> Float
rectangleArea ab = a * b
```

Classical geometry. Although there are a couple of things to note. Since a cube is a special case of a hexahedron, we have defined the area and volume by treating it as a hexahedron with all its sides equal. We have also defined a helper function called `rectangleArea`, which calculates the area of a rectangle based on the size of its sides. It is very trivial since it is a multiplication. We have used this function in the `cuboidArea` and `cuboidVolume` functions but we have not exported it. This is because we want our module to only show functions to handle these three dimensional objects, we have used `rectangleArea` but we have not exported it.

When we are creating a module, we usually export only the functions that act as a kind of interface to our module so that the implementation is kept hidden. If someone uses our `Geometry` module, we don't have to worry about functions the functions we don't export. We can decide to change those functions completely or to eliminate them in exchange for a new version (we could eliminate `rectangleArea` and use `*`) and nobody would notice since we are not exporting them.

To use our modules we simply use:

```
import geometry
```

Although `Geometry.hs` must be in the same directory as the program that is using it.

We can also give modules a hierarchical structure. Each module can have any number of submodules and they themselves can have any other number of submodules. We are going to divide the functions of the `Geometry` module into three submodules so that each object has its own module.

First we create a directory called `Geometry`. Keep the G capitalized. Within it will create files `sphere.hs`, `cuboid.hs`, and `cube.hs`. This will be the content of the files:

```
sphere.hs
module Geometry.Sphere
(volume
```

```
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
radius area = 4 * pi * (radius ^ 2)
cuboid.hs
module Geometry.Cuboid
(volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume abc = rectangleArea ab * c

area :: Float -> Float -> Float -> Float
area abc = rectangleArea ab * 2 + rectangleArea ac * 2 + rectangleArea cb * 2

rectangleArea :: Float -> Float -> Float
rectangleArea ab = a * b
cube.hs
module Geometry.Cube
(volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Well! The first one is `Geometry.Sphere`. Note that we have first created a folder called `Geometry` and then and then we have defined the name as `Geometry.Sphere`. We did the same with the hexahedron. Also note that in the three modules we have defined functions with the same names. We can do this because they are separated into different modules. We want to use the `Geometry.Cuboid` functions in `Geometry.Cube` but we cannot just use `import Geometry.Cuboid` since we would import functions with the same name as in `Geometry.Cube`. For this reason we qualify it.

So if we are now in a file that is in the same place as the `Geometry` folder we can use:

```
import Geometry.Sphere
```

And then we can use area and volume and they will give us the area and volume of a sphere. If we want to use two or more modules of these, we have to qualify them so that there are no conflicts with names. We can use something like:

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

Now we can call Sphere.area , Sphere.volume , Cub oid.area , etc. and each will calculate the area or volume of their respective object.

The next time you find yourself writing a module that is very large and have a lot of functions, try to find which functions have a common purpose and then try to put them in the same module. In this way, you will be able to import said module the next time you write a program and require the same functionality.

Chapter VI

Creating our own types and type classes

In earlier chapters we saw some types and classes of Haskell types. In this chapter we will see how to create them ourselves! Did you not expect it?

Introduction to algebraic data types

Until now we have played with many types: Bool , Int , Char , Maybe , etc. But how do we create them? Well, one way is to use the data keyword to define a type. Let's see how the Bool type is defined in the standard library:

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

data means that we are going to define a new data type. The part to the left of the = denotes the type, which is Bool . The part on the right is the **data constructors** . These specify the different values that a type can have. The | can be read as one *or* . So we can read it as: The type Bool can have a value of True or False . Both the type name and the data constructors must be capitalized with the first letter.

In the same way we can think that the type Int is defined as:

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

The first and last data constructors are the minimum and maximum possible value of type Int . It's not actually defined like that, all three dots are there because we have omitted quite a few numbers, so this is for illustration purposes only.

Now let's think about how we would define a figure in Haskell. One way would be to use tuples. A circle could be (43.1, 55.0, 10.4) where the first and second fields are the coordinates of the center of the circle while the third field is the radius. Sounds good, but this would also allow us to define a 3D vector or anything else. A better solution would be to create our own type that represents a shape. Let's say that a figure can only be a circle or a rectangle:

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

What is this? Think about what it looks like. The ` Circle data constructor has three fields that take floating point values. When we create a data constructor, we can optionally add types after it so that these will be the values it contains.

Here, the first two components are the coordinates of the center, while the third is the radius. The Rectangle data constructor has four fields that accept floating point values. The first two represent the coordinates of the upper left corner and the other two represent the coordinates of the lower right.

Now when we talk about fields, we are actually talking about parameters. Data constructors are actually functions that return a value of the type for which they were defined. Let's look at the type declaration of these two data constructors.

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

Well, data constructors are functions like everything else. Who would have thought? We are going to make a function that takes a figure and returns its surface or area:

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

The first notable thing here is the type declaration. Says it takes a shape and returns a floating point value. We cannot write a type declaration like Circle -> Float since Circle is not a type, Shape is. Similarly we cannot declare a function whose type declaration is True -> Int . The next thing we can highlight is that we can use pattern matching with constructors. We've already used pattern matching with constructors before (actually all the time) when we wrap values like [] , False , 5 , only those values don't have fields. We simply write the constructor and then link its fields to names. Since we are interested in radius, we don't really care about the first two values that tell us where the circle is.

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

Well it works! But if we try to display Circle 10 20 5 on a GHCi session we will get an error. This happens because Haskell still doesn't know how to represent our type with a string. Remember that when we try to display a value on the screen, Haskell first executes the show function to get the text representation of a data and then displays it in the terminal. To make our Shape type part of the Show type class we do this:

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

We are not going to worry about referral right now. We will simply say that if we add deriving (Show) to the end of a type declaration, Haskell automatically makes that type part of the Show type class . So now we can do this:

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

Data constructors are functions, so we can map them, partially apply them, or anything else. If we want a list of concentric circles with different radius we can write this:

ghci > map (Circle 10 20) [4 , 5 , 6 , 6]
[Circle 10.0 20.0 4.0 , Circle 10.0 20.0 5.0 , Circle 10.0 20.0 6.0 , Circle 10.0 20.0 6.0]

Our data type is good, but it could be better. We are going to create an intermediate data type that defines a point in two-dimensional space. Then we will use it to make our type more evident.

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

You may have noticed that we have used the same name for the type as for the data constructor. There is nothing special, it is common to use the same name as the type if there is only one data constructor. So now Circle has two fields, one is of the Point type and the other of the Float type . In this way it is easier to understand what each thing is. The same is true for the rectangle. We have to modify our surface function to reflect these changes.

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

The only thing we've changed has been the patterns. We have completely discarded the point in the circle pattern. On the other hand, in the rectangle pattern, we have simply used a nested pattern fit to obtain the coordinates of the points. If we had wanted to make a direct reference to the points for any reason we could have used a pattern *like* .

ghci > surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci > surface (Circle (Point 0 0) 24)
1809.5574

What would a function that displaces a figure look like? It would take a shape, the amount to be shifted on the *x* axis , the amount to be shifted on the

y axis, and would return a new shape with the same dimensions but shifted.

```
nudge :: Shape -> Float -> Float -> Shape
nudge ( Circle ( Point x y ) r ) a b = Circle ( Point ( x + a ) ( y + b )) r
nudge ( Rectangle ( Point x1 y1 ) ( Point x2 y2 )) a b = Rectangle ( Point ( x1 + a ) ( y1 + b )) ( Point ( x2 + a ) ( y2 + b ))
```

Pretty straightforward. We add the quantities to move to the points that represent the position of the figures.

```
ghci > nudge ( Circle ( Point 34 34 ) 10 ) 5 10
Circle ( Point 39.0 44.0 ) 10.0
```

If we don't want to work directly with points, we can create auxiliary functions that create figures of some size in the center of the coordinate axis so that we can later move them.

```
baseCircle :: Float -> Shape
baseCircle r = Circle ( Point 0 0 ) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle ( Point 0 0 ) ( Point width height )
ghci > nudge ( baseRect 40 100 ) 60 23
Rectangle ( Point 60.0 23.0 ) ( Point 100.0 123.0 )
```

Obviously, we can export our data in the modules. To do this, we only have to write the type name together to the exported functions, and then add parentheses containing the data constructors that we want to export, separated by commas. If we want all builders exported data for a certain type we can use .. .

If we wanted to export the functions and types we just created in a module, we could start with this:

```
module Shapes
( Point ( .. )
, Shape ( .. )
, surface
, Nudge
, baseCircle
, baseRect
) where
```

By doing Shape (..) we are exporting all the Shape data constructors , which means that anyone who imports our module can create shapes using the Circle and Rectangle constructors . It would be the same as writing Shape (Rectangle, Circle) .

We could also choose not to export any data constructors for Shape simply by typing Shape in that statement. In this way, whoever imports our module can only create figures using the auxiliary functions baseCircle and baseRect .

Data.Map uses this method. You cannot create a dictionary using Map.Map [(1,2), (3,4)] since the data constructor is not exported. However, we can create a dictionary using helper functions like Map.fromList . Remember, data constructors are simple functions that take fields of type as parameters and return a value of a certain type (such as Shape) as a result. So when we choose not to export them, we are preventing people importing our module from using those functions, but if some other function returns the type we are exporting, we can use them to create our own values of that type.

Not exporting the data constructors of a data type makes it more abstract in the sense that it hides its implementation. However, module users will not be able to use pattern matching on that type.

Registration syntax

Well, we have been given the task of creating a type that describes a person. The information we want to store for each person is: name, surname, age, height, telephone number and the taste of their favorite ice cream. I don't know anything about you, but for me it's everything I need to know about a person. Let's go there!

```
data Person = Person String String Int Float String String deriving ( Show )
```

Voucher. The first field is the name, the second the last name, the third the age and we continue counting. We are going to create a person.

```
ghci > let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci > guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

It seems interesting, but certainly not very readable. What if we want to create a function that gets information separately from a person? A function that gets a person's name, another function that gets their last name, etc. Well, we would have to define them like this:

```
firstName :: Person -> String
firstName ( Person firstname _ _ _ _ _ ) = firstname

lastName :: Person -> String
lastName ( Person _ lastname _ _ _ _ ) = lastname

age :: Person -> Int
age ( Person _ _ age _ _ _ ) = age

height :: Person -> Float
height ( Person _ _ _ height _ _ ) = height
```

```
phoneNumber :: Person -> String
phoneNumber ( Person _ _ _ _ number _ ) = number

flavor :: Person -> String
flavor ( Person _ _ _ _ _ flavor ) = flavor
```

Wow! I really didn't have fun writing this. Other than this method being messy and a little BORED to write, it works.

```
ghci > let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci > firstName guy
"Buddy"
ghci > height guy
184.2
ghci > flavor guy
"Chocolate"
```

Now is when you think: there must be a better method. Well no, I'm so sorry.

I was kidding: P Yes there is. The creators of Haskell were very smart and anticipated this scenario. They included an alternative method of defining data types. This is how we could achieve the same functionality with the registry syntax.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving ( Show )
```

Instead of naming the fields one after the other separated by spaces, we use a pair of curly braces. Inside, we first write the name of a field, for example firstName, and then we write double dots :: (also known as *Paamayim Nekudotayim* xD) and then specify the type. The resulting data type is exactly the same. The main difference is that in this way functions are created that obtain those fields of the data type. By using registration syntax with this data type, Haskell automatically creates these functions: firstName , lastName , age , height , phoneNumber and flavor .

```
ghci > : t flavor
flavor :: Person -> String
ghci > : t firstName
firstName :: Person -> String
```

There is another benefit when we use the registry syntax. When we derive Show for a type, it will display the data differently if we use the registration syntax to define and instantiate the type. Suppose we have a type that represents a car. We want to keep a record of the company that made it, the

model name and its years of production. Look.

```
data Car = Car String String Int deriving ( Show )
ghci > Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

If we define it using the registry syntax, we can create a new car like this:

```
data Car = Car { company :: String , model :: String , year :: Int } deriving ( Show )
ghci > Car { company = "Ford" , model = "Mustang" , year = 1967 }
Car { company = "Ford" , model = "Mustang" , year = 1967 }
```

When we create a new car, it is not necessary to put the fields in the proper order while we put them all. But if we do not use the registration syntax we must specify them in their correct order.

Use the registration syntax when a constructor has multiple fields and it is not obvious which field each is. If we define the type of a 3D vector as data Vector = Vector Int Int Int , it is quite obvious that those fields are the components of the vector. However, in our Person and Car types , it is not so obvious and the use of this syntax benefits us a lot.

Type parameters

A data constructor can take some values as parameters and produce a new value. For example, the Car constructor takes three values and produces a value of type car. Similarly, a type **constructor** can take types as parameters and produce new types. This may seem a little recursive at first, but it's not complicated at all. If you have used *C ++* templates it will be familiar to you. To get a clear picture of how type parameters actually work, let's look at an example of how a type we already know is implemented.

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

The a is a type parameter. Because there is a type parameter involved in this definition, we call Maybe a type constructor. Depending on what we want this type to contain when a value is not Nothing , this type can end up producing types like Maybe Int , Maybe Car , Maybe String , etc. No value can have a type that is simply Maybe , since that is not a type by itself, it is a type constructor. For it to be a real type that any value can have, it has to have all the type parameters defined.

If we pass Char as a type parameter to Maybe , we will get the Maybe Char type . For example, the value Just 'a' has the type Maybe Char .

You may not know it, but we used a type that had a type parameter before we started using the Maybe type . That guy is the list guy. Although there is a little syntactic decoration, the list type takes a parameter to produce a specific type. Values can have a type [Int] , a type [Char] , [[String]] , etc. but there cannot be a value whose type is simply [] .

Let's play around with the Maybe guy a little bit .

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

Type parameters are useful since they allow us to create different types depending on the type we want to store in our data types (redundancy notwithstanding). When we do : t Just "Haha" the type inference engine deduces that the type must be Maybe [Char] , since the a in Just a is a string, then the a in Maybe a must also be a string.

As you may have seen the type of Nothing is Maybe a . Its type is polymorphic. If a function requires a Maybe Int as a parameter we can pass it a Nothing since it does not contain any value. The Maybe a type can behave like a Maybe Int , in the same way that 5 can behave like an Int or a Double . Similarly, the type of empty lists is [a] . An empty list can behave like any other list. So we can do things like [1,2,3] ++ [] and ["ha", "ha", "ha"] ++ [] .

The use of type parameters can benefit us, but only in cases that make sense. We usually use them when our data type will work the same regardless of the data type it contains, just like our Maybe a . If our type is like some kind of box, it is a good place to use type parameters. We could change our Car type from:

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

TO:

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

But does it have any benefit? The answer is: probably not, since in the end we will end up writing functions that only work with the Car String String Int type . For example, given the first definition of Car , we could create a function that would show the properties of a car with a little text:

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

A very nice feature! The type declaration is simple and works perfectly. Now what would it be like if Car were actually Car a b c ?

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

We have to force the function to take a Car of the type (Show a) => Car String String a . We can see how the type definition is much more complicated and the only benefit we have obtained is that we can use any type that is an instance of the Show type class as parameter c .

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

At the moment of truth, we would end up using Car String String Int most of the time and we would realize that parameterizing the Car type doesn't really

matter. We usually use type parameters when the type that is contained within the data type is not really important when working with it. A list of things is a list of things and no matter what those things are, it will work the same. If we want to add a list of numbers, we can later specify in the addition function itself that we specifically want a list of numbers. The same goes for Maybe . Maybe represents the choice of whether or not to have a value. It doesn't really matter what type that value is.

Another example of a parameterized type that we already know is the Map k v type of Data.Map . k is the type for the dictionary keys while v is the type of the values. This is a good example where type parameters are useful. Having the parameterized dictionaries allow us to associate any type with any other type, as long as the type key is of the Ord type class . If we were defining the dictionary type we could add a class constraint in the definition:

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

However, there is a consensus in the Haskell world that **we should never add class constraints to type definitions** . Why? Well, because it doesn't benefit us much, but in the end we ended up writing more class restrictions, even if we don't need them. Whether or not we can put the Ord k class constraint on the Map k v type definition , we will still have to put the class constraint on functions that assume the keys are sortable. But if we don't put the constraint in the type definition, we don't have to put (Ord k) => in the type declaration of the functions that don't care if the key can be sortable or not. An example of this would be the toList function that simply converts a dictionary to an association list. Its type declaration is toList :: Map k a -> [(k, a)] . If Map k v had a constraint in its declaration, the type of toList should have been toList :: (Ord k) => Map k a -> [(k, a)] even if the function does not need to compare any keys.

So don't put class restrictions on type declarations even if it makes sense, because in the end you're going to have to put them on function type declarations anyway.

We are going to implement a type for 3D vectors and create some operations with them. We are going to use a parameterized type since, although it will normally contain numbers, we want it to support several types of them.

```
data Vector a = Vector a a a deriving ( Show )

vplus :: ( Num t ) => Vector t -> Vector t -> Vector t
( Vector i j k ) ` VPlus ` ( Vector l m n ) = Vector ( i + l ) ( j + m ) ( k + n )
```

```
vectMult :: ( Num t ) => Vector t -> t -> Vector t
( Vector i j k ) ` vectMult ` m = vector ( i * m ) ( j * m ) ( k * m )

scalarMult :: ( Num t ) => Vector t -> Vector t -> t
( Vector i j k ) ` scalarMult ` ( Vector l m n ) = i * l + j * m + k * n
```

vplus is used to add two vectors. Vectors are added simply by adding their corresponding components. scalarMult calculates the scalar product of two vectors and vectMult calculates the product of a vector and a scalar. These functions can operate as types Vector Int , Vector Integer , Vector Float or anything else while to the vector to a member of class types Num . Also, if you look at the type declaration of these functions, you will see that they can only operate with vectors of the same type and the numbers involved (as in vectMult) must also be of the same type as that contained in the vectors. Note that we have not put a Num class constraint on the Vector type declaration , as we should have repeated this on the function declarations as well.

Again, it is very important to distinguish between data constructors and type constructors. When we declare a data type, the part before the = is the type constructor, while the part after it (possibly separated by |) is the data constructors. Giving a function the type Vector t t t -> Vector t t t -> t would be incorrect since we have used types in the declaration and the vector type constructor takes only one parameter, while the data constructor takes three. Let's play around with the vectors a bit:

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

Derived instances

A type can be an **instance** of that class if it supports that behavior. Example: The Int type is an instance of the Eq class , since the Eq type class defines the

behavior of things that can be equated. And since integers can be equated, Int is part of the Eq class . The real utility is in the functions that act as an interface for Eq , which are == and / = . If a type is part of the Eq class , we can use functions like == with values of that type. For this reason, expressions like 4 == 4 and "foo" / = "bar" are correct.

We also mention that type classes are often confused with language classes like Java, Python, C ++ and so on, which later puzzles people. In these languages, classes are like a model from which we can create objects that contain a state and can do some actions. Type classes are more like interfaces. We do not create instances from the interfaces. Instead, we first create our data type and then think what it might behave like. If it can behave like something that can be equated, we make it a member of the Eq class . If it can be put in any order, we make it a member of the Ord class .

Later we will see how we can manually make our types an instance of a type class by implementing the functions that it defines. But now, let's see how Haskell can automatically make our types belong to one of the following classes: Eq , Ord , Enum , Bounded , Show and Read . Haskell can derive the behavior of our types in these contexts if we use the deriving keyword when we define them.

Consider the following data type:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Describe a person. Let's assume that no person has the same combination of name, surname and age. Now, if we have two people registered, does it make sense to know if these two records belong to the same person? It looks like it is. We can compare them by equality and see if they are the same or not. For this reason it makes sense for this type to be a member of the Eq type class . We derive the instance:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving ( Eq )
```

When we derive an instance of Eq for a type and then try to compare two values of that type using == or / = , Haskell will check if the type constructors

match (although there is only one type constructor here) and then check if all the fields of that constructor match using the = operator for each pair of fields. We only have to take one thing into account, all the fields of the type must also be members of the Eq type class . Since String and Int are already members, there is no problem. Let's check our Eq instance .

```
ghci > let mikeD = Person { firstName = "Michael" , lastName = "Diamond" , age = 43 }
ghci > let adRock = Person { firstName = "Adam" , lastName = "Horovitz" , age = 41 }
ghci > let mca = Person { firstName = "Adam" , lastName = "Yauch" , age = 44 }
ghci > mca == adRock
False
ghci > mikeD == adRock
False
ghci > mikeD == mikeD
True
ghci > mikeD == Person { firstName = "Michael" , lastName = "Diamond" , age = 43 }
True
```

Since Person is now part of the Eq class , we can use it as a in functions that have a class constraint of type Eq a in their declaration, such as elem .

```
ghci > let beastieBoys = [ mca , adRock , mikeD ]
ghci > miked ` elem ` Beastie Boys
True
```

The Show and Read type classes are for things that can be converted to or from strings, respectively. As with Eq , if a type constructor has fields, its type must be a member of the ` Show or Read class if we want it to be part of these classes as well.

We are going to make our Person data type also part of the Show and Read classes .

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving ( Eq , Show , Read )
```

Now we can show a person through the terminal.

```
ghci > let mikeD = Person { firstName = "Michael" , lastName = "Diamond" , age = 43 }
ghci > mikeD
Person { firstName = "Michael" , lastName = "Diamond" , age = 43 }
ghci > "mikeD is:" ++ show mikeD
"mikeD is: Person {firstName = \" Michael \ " , lastName = \" Diamond \ " , age = 43}"
```

If we had tried to display a person in the terminal before making the Person type part of the Show class , Haskell would have complained, telling us that he doesn't know how to represent a person with a string. But now that we

have derived the Show class, you already know how to do it.

Read is practically the reverse class of Show . Show serves to convert our type to a string, Read serves to convert a string to our type. Although remember that when you use the read function you have to use an explicit type annotation to tell Haskell what type we want as a result. If we don't explicitly put the type we want, Haskell won't know what type we want.

ghci > read "Person {firstName = \" Michael \ " , lastName = \" Diamond \ " , age = 43}" :: Person
Person { firstName = "Michael" , lastName = "Diamond" , age = 43 }

There is no need to use an explicit type annotation in case we use the result of the read function so that Haskell can infer the type.

ghci > read "Person {firstName = \" Michael \ " , lastName = \" Diamond \ " , age = 43}" == mikeD
True

We can also read parameterized types, but we have to specify all the parameters of the type. So we can't read "Just 't'" :: Maybe a but we can read "Just 't'" :: Maybe Char .

We can derive instances for the Ord type class , which is for types whose values can be ordered. If we compare two values of the same type that were defined using different constructors, the value whose constructor was defined first is considered less than the other. For example, the Bool type can have False or True values . In order to see how it behaves when compared, we can think that it is implemented in this way:

data Bool = False | True deriving (Ord)

Since the False value is defined first and the True value is defined later, we can consider that True is greater than False .

ghci> True *compare* False GT ghci> True> False True ghci> True <False False

In the Maybe a type , the Nothing data constructor is defined before the Just constructor , so a Nothing value is always smaller than any Just something value , even if that something is less than a trillion trillion. But if we compare two Just values , then what is inside it is compared.

```
ghci > Nothing < Just 100
True
ghci > Nothing > Just ( - 49999 )
False
ghci > Just three ` compare ` Just 2
GT
ghci > Just 100 > Just 50
True
```

We can't do something like Just (* 3) > Just (* 2) , since (* 3) and (* 2) are functions, which have no Ord instance defined .

We can easily use algebraic data types to create enumerations, and the Enum and Bounded type classes will help us do that. Consider the following data type:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Since no data constructor has parameters, we can make it a member of the Enum type class . The Enum class is for things that have a predecessor and successor. We can also make it a member of the Bounded type class , which is for things that have a minimum possible value and maximum possible value. Now that we get, let's make this type have an instance for all the different types of derivable types we've seen and we'll see what we can do with it.

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
        deriving ( Eq , Ord , Show , Read , Bounded , Enum )
```

Since it is part of the Show and Read type classes , we can convert values of this type to and from strings.

```
ghci > Wednesday
Wednesday
ghci > show Wednesday
"Wednesday"
ghci > read "Saturday" :: Day
Saturday
```

Since it is part of the Eq and Ord type classes , we can compare or equate days.

```
ghci > Saturday == Sunday
False
ghci > Saturday == Saturday
True
ghci > Saturday > Friday
True
ghci > Monday ` compare ' Wednesday
LT
```

It is also part of Bounded , so we can get the lowest day or the highest day.

ghci > minBound :: Day
Monday
ghci > maxBound :: Day
Sunday

It is also an instance of the Enum class . We can get the predecessor and successor of a day and we can even create range lists with them.

ghci > succ Monday
Tuesday
ghci > pred Saturday
Friday
ghci > [Thursday .. Sunday]
[Thursday , Friday , Saturday , Sunday]
ghci > [minBound .. maxBound] :: [Day]
[Monday , Tuesday , Wednesday , Thursday , Friday , Saturday , Sunday]

Pretty impressive.

Type synonyms

Earlier we mentioned that the [Char] and String types were equivalent and interchangeable. This is implemented with **type synonyms** . Type synonyms don't do anything on their own, they just give some type a different name, so that it gets some meaning to someone reading our code or documentation. Here is how you define the standard String library as a synonym for [Char] .

type String = [Char]

We just entered the type keyword . This keyword might mislead some as we are not actually doing anything new (we are doing it with the data keyword). We are simply giving synonyms to a type that already exists.

If we make a function that converts a string to uppercase and call it toUpperString or something similar, we can give it a type declaration like toUpperString :: [Char] -> [Char] or toUpperString :: String -> String . Both are essentially the same, only the latter is more readable.

When we were talking about the Data.Map module , we first presented a represented phone book with an association list and then turned it into a dictionary. As we already know, an association list is nothing more than a list of key-value pairs. Let's go back to see the list we had.

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

We see that the phoneBook type is [(String, String)] . This tells us that it is an association list that associates strings with strings, but nothing else. We are going to create a type synonym to convey some more information in the type declaration.

```
type PhoneBook = [( String , String )]
```

Now the type declaration of our phoneBook function would be phoneBook :: PhoneBook . We are going to make a type synonym for strings too.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [( Name , PhoneNumber )]
```

Giving a synonym to the String type is something that Haskell programmers usually do when they want to convey some more information about the role of strings in their functions and what they represent.

So now, when we implement a function that takes the name and phone number and looks to see if that combination is in our phone book, we can give it a very descriptive type statement:

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = ( name , pnumber ) ` elem ` pbook
```

If we decide not to use type synonyms, our function would have the type declaration String -> String -> [(String, String)] -> Bool . In this case, the type declaration using type synonyms is much clearer and easier to understand. However, you should not abuse them. We use type synonyms either to indicate that it represents a type that already exists in our functions (and thus our type statements become the best documentation) or when something has a very long type that is repeated a lot (like [(String, String)]) and has a specific meaning for us.

Type synonyms can also be parameterized. If we want a type that represents association lists but we also want it to be general enough to use any key type and value, we can use this:

type AssocList k v = [(k , v)]

With this, a function that takes a value per key in an association list can have the type (Eq k) => k -> AssocList k v -> Maybe v . AssocList is a type constructor that takes two types and produces a specific type, like AssocList Int String for example.

Note

When we talk about specific types we are referring to fully applied types, such as Map Int String . Sometimes the boys and I say Maybe is a type, but we don't want to mean that, since any idiot knows Maybe is a type constructor. When I apply an extra type to Maybe , like Maybe String , then I have a specific type. You know, values can only have types that are specific types. Concluding, live fast, love a lot and do not let anyone tease you.

In the same way that we can partially apply functions to get new functions, we can partially apply type parameters and get new type constructors. In the same way that we call functions with minus parameters to get new functions, we can specify a type constructor with minus parameters and get a partially applied type constructor. If we want a type that represents a dictionary (from Data.Map) that associates integers with anything else, we can use this:

type IntMap v = Map Int v

Or else this:

type IntMap = Map Int

Either way, the IntMap type constructor will take a parameter and that will be the type with which the integers will be associated.

Note

If you are going to try to implement this, you will surely import the Data.Map module in a qualified way . When you do a qualified import, the type constructors must also be preceded by the module name. So you have to write something like type IntMap = Map.Map Int .

Make sure you really understand the difference between type constructors and data constructors. Just because we've created a synonym called IntMap or AssocList doesn't mean we can do things like AssocList [(1,2), (4,5), (7,9)] . All it means is that we can refer to that guy using different names. We can do [(1,2), (3,5), (8,9)] :: AssocList Int Int , which will make the numbers inside

assume the type Int , but we can continue using this list as if it were a list that will house pairs of integers. Type synonyms (and types in general) can only be used in the Haskell portion dedicated to types. We will be in this portion of Haskell when we are defining new types (both in data declarations and in type declarations) or when we are placed after a :: . :: is used only for declarations or type annotations.

Another interesting data type that takes two types as a parameter is the Either a b type . This is how it is defined more or less:

data Either a b = Left a | Right b deriving (Eq , Ord , Read , Show)

It has two data constructors. If Left is used , then it contains data of type a and if Right is used it contains data of type b . We can use this type to encapsulate a value of one type or another and thus obtain a value of the type Either a b . Normally we will use a pattern adjustment with both Left and Right , and we will differentiate according to one or the other.

ghci > Right 20
Right 20
ghci > Left "w00t"
Left "w00t"
ghci > : t Right 'a'
Right 'a' :: Either a Char
ghci > : t Left True
Left True :: Either Bool b

So far we have seen that Maybe a is used to represent calculation results that may or may not have failed. But sometimes Maybe a is not good enough since Nothing only informs us that something has gone wrong. This is fine for functions that can only fail in one way or if we are not interested in knowing why and how they have failed. A search in a Data.Map only fails when the key we are looking for cannot be found in the dictionary, so we know exactly what happened. However, when we are interested in how or why something has failed, we usually use the Either type a b as a result , where a is some kind of type that can tell us something about a possible failure, and b is the type of a calculation. satisfactory. Therefore, errors use the Left data constructor while the results use Right .

An example: An institute has lockers so that its students have a place to keep their *Guns'n'Roses* posters . Each box office has a combination. When a student wants a new locker, he tells the locker supervisor what locker number

he wants and he gives him a code for that locker. However, if someone is already using the locker, they cannot tell you the code and they have to choose a different locker. We will use a Data.Map dictionary to represent the lockers. It will associate the box office number with pairs containing whether the box office is in use or not and the box office code.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving ( Show , Eq )

type Code = String

type LockerMap = Map . Map Int ( LockerState , Code )
```

Pretty simple. We have created a new data type to represent whether a locker is free or not, and we have created a synonym to represent the code of a locker. Also created another synonym for the type that associates the box office numbers with the status and code pairs. Now, let's do a function that looks up a box office number in the dictionary. We are going to use the Either String Code type to represent the result, since our search can fail in two ways: the box office has already been taken, in which case we say who owns it or if there is no box office with that number. If the search fails, we will use a string to get why.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map . lookup lockerNumber map of
        Nothing -> Left $ "Locker number" ++ show lockerNumber ++ "doesn't exist!"
        Just ( state , code ) -> if state / = Taken
                                    then Right code
                                    else Left $ "Locker" ++ show lockerNumber ++ "is already
taken!"
```

We do a normal search in a dictionary. If we get Nothing , we return a value with type Left String that says that box office does not exist. If we find it, we do an additional check to see if the locker is free. If it is not, we return a Left saying that the box office has been taken. If it is, we return a value of the Right Code type , which we will give to the student. It is actually a Right String , although we have created a synonym to add a little more information in the type declaration. Here is an example dictionary:

```
lockers :: LockerMap
```

```
lockers = Map . fromList
    [( 100 , ( Taken , "ZD39I" ))
    , ( 101 , ( Free , "JAH3I" ))
    , ( 103 , ( Free , "IQSA9" ))
    , ( 105 , ( Free , "QOTSA" ))
    , ( 109 , ( Taken , "893JJ" ))
    , ( 110 , ( Taken , "99292" ))
    ]
```

Let's find the code for a few lockers:

```
ghci > lockerLookup 101 lockers
Right "JAH3I"
ghci > lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci > lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci > lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci > lockerLookup 105 lockers
Right "QOTSA"
```

We could have used the Maybe a type to represent the result but then we wouldn't know the reason why we can't get the code. Now, we have information about the failure in our result type.

Recursive data structures

As we have already seen, a constructor of an algebraic data type can have (or not have) several fields and each of these must be a specific type. With this in mind, we can create types whose constructor fields are the type itself. In this way, we can create recursive data structures, in which a value of a certain type contains values of that same type, which will continue to contain values of the same type and so on.

Think of the list [5] . It is the same as 5: [] . To the left of : there is a value, and to the right is a list. In this case, an empty list. What would happen to the list [4,5] ? Well it's the same as 4: (5: []) . If we look first : we see that also has an element to your left and on your right list (5: []) . The same is true for list 3: (4: (5: 6: [])) , which could also be written as 3: 4: 5: 6: [] (since : is associative from the right) or [3, 4,5,6] .

We can say that a list is either an empty list or an element joined with a : to

another list (which can be an empty list or not).

Let's use algebraic data types to implement our own list!

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

It reads the same way our list definition read in a previous paragraph. It is either an empty list or a combination of one item and another list. If you are confused with this, you may find it easier to understand it with the registry syntax:

data List a = Empty | Cons { listHead :: a , listTail :: List a } deriving (Show , Read , Eq , Ord)

You may also be confused with the Cons constructor . Cons is another way of saying : . Indeed, in the lists : a constructor that takes a value and a list Returns a list. In other words, it has two fields. One is of type a and the other is of type [a] .

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

If we had called our builder infix could see better as simply : . Empty is like [] and 4 `Cons` (5` Cons` Empty) is like 4: (5: []) .

We can define functions that are automatically infix if we name them only with special characters. We can do the same with constructors, since they are simply functions that return a specific data type. Watch this:

infixr 5 : -:
data List a = Empty | a : -: (List a) deriving (Show , Read , Eq , Ord)

First of all, we see that there is a new syntactic construction, an infix statement. When we define functions as operators, we can use this construct to give them a certain behavior (although we are not obliged to do so). In this way we define the order of precedence of an operator and whether associative from the left or from the right. For example, * is infixl 7 * and + is infixl 6 + . This means that both are associative from the left so that (4 * 3 * 2) is (4 * 3) * 2) but * has an order of precedence greater than + , so 5 * 4 + 3 is equivalent to (5 * 4) + 3 .

Either way, we end up writing to : -: (List a) instead of `` Cons a (List a) ".

Now we can write the lists like this:

```
ghci > 3 : -: 4 : -: 5 : -: Empty
( : -: ) 3 (( : -: ) 4 (( : -: ) 5 Empty ))
ghci > let a = 3 : -: 4 : -: 5 : -: Empty
ghci > 100 : -: a
( : -: ) 100 (( : -: ) 3 (( : -: ) 4 (( : -: ) 5 Empty )))
```

Haskell will continue showing the cosntructor as a prefix function when we derive Show , for this reason the brackets appear around the constructor (remember that 4 + 3 is the same as (+) 4 3).

We are going to create a function that joins two of our lists. This is how the ++ function is defined for normal lists:

```
infixr 5 ++
( ++ ) :: [ a ] -> [ a ] -> [ a ]
[] ++ ys = ys
( x : xs ) ++ ys = x : ( xs ++ ys )
```

So we copy this definition and apply it to our lists:

```
infixr 5. ++
( . ++ ) :: List a -> List a -> List a
Empty . ++ ys = ys
( x : -: xs ) . ++ ys = x : -: ( xs . ++ ys )
```

And this is how it works:

```
ghci > let a = 3 : -: 4 : -: 5 : -: Empty
ghci > let b = 6 : -: 7 : -: Empty
ghci > a . ++ b
( : -: ) 3 (( : -: ) 4 (( : -: ) 5 (( : -: ) 6 (( : -: ) 7 Empty ))))
```

Well. If you want you can implement all the functions that operate with lists with our type of lists.

Notice that we have used a pattern adjustment (x : -: xs) . This works since pattern matching actually works by adjusting constructors. We can adjust a pattern : -: because it is a constructor of our type in the same way that : it is a constructor of standard lists. The same is true for [] . Since pattern matching works (only) with data constructors, we can adjust patterns like normal prefix constructors, infix constructors, or things like 8 or 'a' , which are both number and character type constructors after all.

We are going to implement a binary search tree. If you are not familiar with

the binary search trees of other languages like *C* , here is an explanation of what they are: one element points to two other elements, one is on the left and one on the right. The item on the left is smaller and the second item is larger. Each of these two elements can point to two other elements (or to one or neither). Indeed, each element has its own sub-trees. The good thing about binary search trees is that we know that all the elements that are in the sub-tree on the left of, 5, for example, are less than 5. The elements that are in the sub-tree on the right are greater. So if we are looking for element 8 in our tree, we start comparing it with 5, since we see that it is less than 5, we go to the sub-tree on the right. Now we would be at 7, as it is less than 8 we would continue to the right. In this way we would find the element in three steps. If we were using a list (or an unbalanced tree), it would have taken us about 7 steps to find 8.

The Data.Set and Data.Map sets and dictionary are implementing using trees, only instead of search binary trees, they use balanced search binary trees, so they are always balanced. Now we will simply implement normal binary search trees.

Let's say that: a tree is either an empty tree or an element that contains an element and two other trees. It looks like it will fit perfectly with algebraic data types.

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

Voucher. Instead of manually building a tree, let's create a function that takes an element and a tree and inserts that element into its proper position within the tree. We do this by comparing the element that we want to insert with the root of the tree and if it is less, we go to the left and if not to the right. We do the same for each next node until we reach an empty tree. When we do that we simply insert the element instead of the empty tree.

In languages like *C* , we perform this task by modifying the pointers and values in the tree. In Haskell, we cannot modify our tree, so we have to create a new sub-tree every time we decide if we go to the right or to the left and at the end the insert function returns a completely new tree, since Haskell does not have the pointer concept. So the type declaration of our function will be something like a -> Tree a - > Tree a . It takes an element and a tree and returns a new tree that has that element inside. It may seem inefficient, but Hasekell's lazy evaluation already takes care of it.

Here you have two functions. One of them is an auxiliary function to create a unitary tree (that only contains an element) and the other is a function that inserts elements in a tree.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

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

The singleton function is a quick way to create a tree containing one element and two empty sub-trees. In the insert function, we have the base case as the first pattern. If we have reached an empty sub-tree, this means that we are where we wanted and instead of an empty tree, we want a unitary tree that contains the element to insert. If we are not inserting the element into an empty tree we have to check several things. First, if the element we are going to insert is the same as the root of the sub-tree, we simply return the tree as it was. If it is less, we return a tree that has the same root, the same right sub-tree but instead of its left sub-tree, we put the tree that will contain this element. The same is true (but in the opposite direction) for values that are greater than the root element.

Next we are going to create a function that checks if an element belongs to a tree. Let's first define the base case. If we are looking for an element in an empty tree, obviously the element is not there. Okay, notice this is basically the same as the base case of list search: if we are looking for an item in an empty list, obviously the item is not there. Anyway, if we are not looking for the item in an empty tree, then we have to do several checks. If the element we are looking for is the root element, great! What if it is not? Well, we have the advantage that we know that all the elements less than the root are in the left sub-tree. So if the element we are looking for is less than the root, we check if the element is in the left sub-tree. If it is greater, we check the right sub-tree.

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

Let's have fun with our trees! Instead of manually building a tree (although we could), we will use a fold to build a tree from a list. Remember, almost anything that loops through an item-by-item list and returns some sort of value can be implemented with a fold. We will start with an empty tree and then we will go through the list from the right and we will insert elements into our accumulator tree.

```
ghci > let nums = [ 8 , 6 , 4 , 1 , 7 , 3 , 5 ]
ghci > let numsTree = foldr treeInsert EmptyTree nums
ghci > numsTree
Node 5 ( Node 3 ( Node 1 EmptyTree EmptyTree ) ( Node 4 EmptyTree EmptyTree )) ( Node 7 (
Node 6 EmptyTree EmptyTree ) ( Node 8 EmptyTree EmptyTree ))
```

In this foldr , treeInsert is the fold function (it takes a tree and an item from the list and produces a new tree) and EmptyTree is the initial value. Of course, nums is the list we are folding.

The tree displayed by the console is not very readable, but if we try, we can decipher its structure. We see that the root node is 5 and then it has two sub-trees, one that has a root element of 3, and another one of 7.

```
ghci > 8 ` treeElem ` numsTree
True
ghci > 100 ` treeElem ` numsTree
False
ghci > 1 ` treeElem ` numsTree
True
ghci > 10 ` treeElem ` numsTree
False
```

Let's check that the relevance of an element to a tree works perfectly. Great.

As you can see the algebraic data types in Hasekll are a very interesting concept as well as powerful. We can use them from to represent boolean values to enumerations of the days of the week, and even binary search trees.

Type classes step by step (2nd part)

So far we have learned to use some standard Haskell type classes and have seen which types are members of them. We've also learned how to automatically create instances of our types for standard type classes, asking Haskell to derive them for us. In this section we will see how we can create our own type classes and how to create type instances for them by hand.

A little reminder about type classes: type classes are like interfaces. A type class defines a behavior (such as compare by equality, compare by order, an enumeration, etc.), and then certain types can behave in a manner to the instance of that type class. The behavior of a type class is achieved by defining functions or simply by defining types that we will later implement. So when we say that a type is an instance of a type class, we are saying that we can use the functions of that type class with that type.

Type classes have nothing to do with *Java* or *Pyhton classes* . This tends to confuse a lot of people, so I would like you to forget right now everything you know about classes in the imperative languages.

For example, the Eq type class is for things that can be matched. Define the == and / = functions . If we have a type (let's say, Car) and comparing two cars with the == function makes sense, then it makes sense for Car to be an instance of Eq .

This is how the Eq class is defined in Prelude :

```
class Eq a where
    ( == ) :: a -> a -> Bool
    ( / = ) :: a -> a -> Bool
    x == y = not ( x / = y )
    x / = y = not ( x == y )
```

Stop, stop, atlo! There is a lot of syntax and weird words there! Don't worry, everything will be clear in a second. First of all, when we write class Eq to where it means we are defining a new type class and it is going to be called Eq . The a is the type variable and means that a will represent the type that we will shortly instantiate Eq . It does not have to be called to , in fact has neither to be a single letter, should only be a word in lowercase. Then we define various functions. It is not mandatory to implement the bodies of the functions, we only have to specify the type declarations of the functions.

Note

There are people who will understand this better if we write something like class Eq comparable where and then define the type of the functions as (==) :: comparable -> comparable -> Bool .

Anyway, we have implemented the body of the functions defined by Eq , only we have implemented them in terms of mutual recursion. We say that two instances of class Eq are equal if they are not unequal and they are

unequal and they are not equal. Actually we didn't have to have done it, but soon we will see how it helps us.

Note

If we have a class Eq a where and we define a type declaration inside the class as (==) :: a -> -a -> Bool , then when we examine the type of that function we will obtain (Eq a) => a -> to -> Bool .

So we already have a class, what can we do with it? Well not much. But once we start declaring instances for that class, we will start to get some useful functionality. Look at this guy:

```
data TrafficLight = Red | Yellow | Green
```

Defines the states of a traffic light. Note that we have not derived any instance, since we are going to write them by hand, although we could have derived them for the Eq and Show classes . Here is how we create the instance for the Eq class .

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

We did this using the instance keyword . So class is to define new type classes and instance to make our types have an instance for a certain type class. When we were defining Eq we wrote class Eq to where and said that a would represent the type that we instantiated after. We can see it clearly now, since when we are writing an instance, we write instance Eq TrafficLight where . We have replaced the a with the current type.

Since == was defined in the class definition in terms of / = and vice versa, we only have to overwrite one of them in the instance statement. This is called the minimum complete definition of a type class, or in other words, the minimum number of functions that we have to implement so that our type belongs to a certain type class. To fill the minimum complete definition of Eq , we have to overwrite either == or / = . If Eq had been defined as:

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

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

We should have implemented both functions when creating an instance, since Hasekell would know how those functions are related. Thus, the minimum complete definition would be both == and / = .

As you have seen we have implemented == using pattern matching. Since there are many more cases where two semaphores are not in the same state, we specify for which they are equal and then we use a pattern that fits any case that is not any of the above to say that they are not equal.

We will also create an instance for Show . To satisfy the minimum complete definition of Show , we just have to implement the show function , which takes a value and converts it to a string.

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Once again we have used pattern matching to achieve our goals. Let's see it in action:

```
ghci > Red == Red
True
ghci > Red == Yellow
False
ghci > Red ` elem ` [ Red , Yellow , Green ]
True
ghci > [ Red , Yellow , Green ]
[ Red light , Yellow light , Green light ]
```

Perfect. We could have derived Eq and it would have had the same effect. However, deriving Show would have directly represented the constructors as strings. But if we want the lights to appear as "Red light" we have to create this instance by hand.

We can also create type classes that are subclasses of other type classes. The Num class declaration is a bit long, but here is the beginning:

```
class ( Eq a ) => Num a where
    ...
```

As we have already mentioned above, there are a lot of places where we can put class restrictions. This is the same as writing class Num a where , only we say that our type a must be an instance of Eq . Basically we say that you must

create the Eq instance of a type before it is part of the Num class . Before a type can be considered a number, it makes sense that we can determine whether the values of a type can be matched or not. This is all there is to know about subclasses as they are simply class constraints within a class definition. When we define functions in the declaration of a class or in the definition of an instance, we can assume that a is part of the Eq class so we can use == with the values of that type.

But how are instances of the Maybe or lists type created? What makes Maybe different from, say, TrafficLight is that Maybe is not itself a specific type, it is a type constructor that takes a parameter (like Char or anything else) to produce a specific type. Let's take a look at the Eq class again:

```
class Eq a where
    ( == ) :: a -> a -> Bool
    ( / = ) :: a -> a -> Bool
    x == y = not ( x / = y )
    x / = y = not ( x == y )
```

From the type declaration, we can see that a is used as a specific type since all the types that appear in a function must be concrete (Remember, you cannot have a function with type a -> Maybe but if a function a -> Maybe a or Maybe Int -> Maybe String). For this reason we cannot do things like:

```
instance Eq Maybe where
    ...
```

Since, as we have seen, a must be a specific type but Maybe is not. It is a type constructor that takes a parameter and produces a specific type. It would be a bit tedious to have to write instance Eq (Maybe Int) ` where , instance Eq (Maybe Char) where , etc. for each type. So we can write it like this:

```
instance Eq ( Maybe m ) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

This is like saying that we want to instantiate Eq for all Maybe types something . In fact, we could have written Maybe something , but we'd rather pick one-letter names to be true to Haskell's style. Here, (Maybe m) plays the role of a in class Eq a where . While Maybe is not a specific type, Maybe m yes. By using a type parameter (m , which is lowercase), we say that we

want all types that are of the form Maybe m , where m is any type that is part of the Eq class .

However, there is a problem with this, can you figure it out? We use == over the contents of Maybe but nobody assures us that what Maybe contains is part of the Eq class . For this reason we have to modify our instance declaration:

```
instance ( Eq m ) => Eq ( Maybe m ) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

We have added a class constraint. With this instance we are saying: We want all types with the form Maybe m to be members of the Eq type class , but only those types where m (what is contained within Maybe) are also members of Eq . This is actually how Haskell would derive this instance.

Most of the time, class constraints in class *declarations* are used to create type classes that are subclasses of other type classes while class constraints in *instance declarations* are used to express the requirements of some kind. For example, we have now expressed that Maybe content should be part of the Eq type class .

When creating an instance, if you see that a type is used as a specific type in the type declaration (like a in a -> a -> Bool), you must add the corresponding type parameters and surround it with parentheses so that you end up having a specific type.

Note

Note that the type you are trying to instantiate for will replace the class declaration parameter. The a of class Eq a where will be replaced with a real type when you create an instance, so mentally try to put the type in the type declaration of the functions. (==) :: Maybe -> Maybe -> Bool doesn't make much sense, but (==) :: (Eq m) => Maybe m -> Maybe m -> Boo yes. But this is simply a way of looking at things, since == will always have the type (==) :: (Eq a) => a -> a -> Bool , regardless of the instances we make.

Oh, one more thing. If you want to see the existing instances of a type class, just do : info YourTypeClass in GHCi. So if we use : info Num will show us what functions are defined in the type class and it will also show us a list with

the types that are part of this class. : info also works with types and type constructors. If we do : info Maybe we will see all the type classes of which it is part. : info also shows you the type of a function. Quite useful.

The Yes-No type class

In JavaScript and other weakly typed languages, you can put almost anything inside an expression. For example, you can do all of the following: if (0) alert ("YEAH!") Else alert ("NO!") , If ("") alert ("YEAH!") Else alert ("NO!") , if (false) alert ("YEAH") else alert ("NO!) , etc. And all of these will show a message saying NO! If we do if (" WHAT ") alert (" YEAH ") else alert (" NO! ") will display " YEAH! " since non-empty strings in JavaScript are considered true values.

Although strict use of Bool for Boolean semantics works best in Haskell, let's try implementing this JavaScript behavior just for fun! Let's start with the class declaration.

```
class YesNo to where
    yesno :: a -> Bool
```

Very simple. The YesNo type class defines a function. This function takes a value of any type that can express some truth value and tells us if it is true or not. Look at the way we use a in the function, a has to be a specific type.

The next thing is to define some instances. For numbers, we assume that (as in JavaScript) any number other than 0 is true and 0 is false.

```
instance Yes No Int where
    yesno 0 = False
    yesno _ = True
```

Empty lists (and by extension strings) are false values, while non-empty lists have a true value.

```
instance YesNo [ a ] where
    yesno [] = False
    yesno _ = True
```

Notice how we have put a type parameter inside to make the list a specific type, although we do not assume anything about what the list contains. What else ... Hmm ... I know! Bool can also contain true and false values and it's

pretty obvious which is which.

```
instance Yes No Bool where
    yesno = id
```

Hey? What is id ? It is simply a standard library function that takes a parameter and returns the same thing, which is the same thing we would have to write here.

We will also instantiate Maybe a .

```
instance YesNo ( Maybe a ) where
    yesno ( Just _ ) = True
    yesno Nothing = False
```

We do not need a class constraint since we do not assume anything about the contents of Maybe . We simply say that it is true if it is a Just value and false if it is Nothing . We still have to write (Maybe a) instead of just Maybe since, if you think about it a bit, a Maybe -> Bool function cannot exist (since Maybe is not a concrete type), while Maybe a -> Bool is Right. Still, it's still great as now, any type Maybe something is part of the class ` YesNo and no matter what something is .

Before we defined a Tree type a to represent the binary search. We can say that an empty tree has a false value while anything else has a true value.

```
instance YesNo ( Tree a ) where
    yesno EmptyTree = False
    yesno _ = True
```

Can the state of a semaphore be a true or false value? Clear. If it's red, you stop. If it is green, you continue. If it is amber? Ehh ... normally I usually accelerate since I live by and for adrenaline.

```
instance Yes No TrafficLight where
    yesno Red = False
    yesno _ = True
```

Great, now we have a few instances, let's play with them:

```
hci > yesno $ length []
False
ghci > yesno "haha"
True
ghci > yesno ""
False
ghci > yesno $ Just 0
True
```

```
ghci > yesno True
True
ghci > yesno EmptyTree
False
ghci > yesno []
False
ghci > yesno [ 0 , 0 , 0 ]
True
ghci > : t yesno
yesno :: ( YesNo a ) => a -> Bool
```

Well it works! We are going to make a function that imitates the behavior of an if statement , but that works with YesNo values .

```
yesnoIf :: ( YesNo y ) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult = if yesno yesnoVal then yesResult else noResult
```

Pretty simple. Take a value with a degree of truth and two other values. If the first value is true, it returns the first value of the other two, otherwise it returns the second.

```
ghci > yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci > yesnoIf [ 2 , 3 , 4 ] "YEAH!" "NO!"
"YEAH!"
ghci > yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci > yesnoIf ( Just 500 ) "YEAH!" "NO!"
"YEAH!"
ghci > yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

The functor type class

So far, we have come across a lot of standard library type classes. We have played with Ord , which is for things that can be ordered. We have seen Eq , which is for things that can be compared. We also saw Show , which serves as an interface for types whose values can be represented as strings. Our good friend Read will be here whenever we need to convert a string to a value of some kind. And now, let's take a look at the Functor type class , which is basically for things that can be mapped. Surely you are thinking about lists right now, since mapping a list is very common in Haskell. And you're right, the list type is a member of the Functor type class .

What better way to learn about the Functor type class than to see how it is implemented? Let's take a look.

```
class Functor f where
    fmap :: ( a -> b ) -> f a -> f b
```

Agree. We have seen that it defines a function, fmap , and does not provide any default implementation for it. The fmap type is interesting. In the type class definitions that we have seen so far, the type variable that has had an important role in the type class has been a specific type, such as a in (==) :: (Eq a) => a -> to -> Bool . But now, f is not a specific type (a type that can have a value, such as Int , Bool, or Maybe String), but a type constructor that takes a type as a parameter. A quick example to remember: Maybe Int is a specific type, but Maybe is a type constructor that takes a type as a parameter. Either way, we have seen that fmap takes a function from one type to another and a functor applied to a type and returns another functor applied with the type otor.

If this sounds a bit confusing to you, don't worry. You will see everything clearer now when we show a few examples. Hmm ... this type statement reminds me of something. If you don't know what the type of map is, it is this: map :: (a -> b) -> [a] -> [b] .

Interesting! It takes a function from one type to another and a list of one type and returns a list of the other type. Friends, I think we just discovered a functor. In fact, map is fmap but it only works with lists. Here's how the lists have an instance for the Functor class .

```
instance Functor [] where
    fmap = map
```

That's! Note that we have not written instance Functor [a] where , since from fmap :: (a -> b) -> f a -> f b we see that f has to be a type constructor that takes a parameter. [a] is already a specific type (a list with any type inside), while [] is a type constructor that takes a parameter and produces things like [Int] , [String] or even [[String]] .

As for lists, fmap is simply map , we get the same result when we use them with lists.

```
map :: ( a -> b ) -> [ a ] -> [ b ]
ghci > fmap ( * 2 ) [ 1 .. 3 ]
[ 2 , 4 , 6 ]
```

```
ghci > map ( * 2 ) [ 1 .. 3 ]
[ 2 , 4 , 6 ]
```

What happens when we make map or fmap on empty lists? Well, of course we obey an empty list. It simply converts an empty list with type [a] to an empty list with type [b] .

The types that can act as a box can be functors. You can think of a list as a box that has an unlimited number of small compartments and may all be empty, or some may be full. So what else has the property of behaving like a box? For example, the type Maybe a . In a way, it is like a box that may either contain nothing, in which case its value will be Nothing , or it may contain something, such as "HAHA" , in which case its value will be Just "HAHA" . Here's how Maybe is a functor:

```
instance Functor Maybe where
    fmap f ( Just x ) = Just ( f x )
    fmap f Nothing = Nothing
```

Again, notice that we have written instance Functor Maybe where instead of instance Functor (Maybe m) where , as we did when using the YesNo class together with Maybe . Functor wants a type constructor that takes a type and not a specific type. If you just replace the f with Maybe , fmap acts like (a -> b) -> Maybe a -> Maybe b for this particular type, which looks good. But if you replace f with (Maybe m) , then it will appear to act like (a -> b) -> Maybe m a -> Maybe m b , which doesn't make any damn sense since Maybe takes a single parameter.

Either way, the implementation of fmap is very simple. If it's an empty value or Nothing , then we simply return Nothing . If we map an empty box we get an empty box. Makes sense. In the same way that if we map an empty list we obtain an empty list. If it is not an empty value, but rather a single value wrapped by Just , then we apply the function to the content of Just .

```
ghci > fmap ( ++ "HEY GUYS IM INSIDE THE JUST" ) ( Just "Something serious." )
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci > fmap ( ++ "HEY GUYS IM INSIDE THE JUST" ) Nothing
Nothing
ghci > fmap ( * 2 ) ( Just 200 )
Just 400
ghci > fmap ( * 2 ) Nothing
Nothing
```

Another thing that can be mapped and therefore can have an instance of Functor is our Tree a type . It can also be seen as a box (contains several or no values) and the Tree type constructor takes exactly one type parameter. If we see the fmap function as if it were a function made exclusively for Tree , its type declaration would be like (a -> b) -> Tree a -> Tree b . We will use recursion with this one. Mapping an empty tree will produce an empty tree. Mapping a non-empty tree will produce a tree in which the function will be applied to the root element and its left and right sub-trees will be the same sub-trees, only they will be mapped with the function.

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f ( Node x leftsub rightsub ) = Node ( f x ) ( fmap f leftsub ) ( fmap f rightsub )
ghci > fmap ( * 2 ) EmptyTree
Emptyree
ghci > fmap ( * 4 ) ( foldr treeInsert EmptyTree [ 5 , 7 , 3 , 2 , 1 , 7 ])
Node 28 ( Node 4 EmptyTree ( Node 8 EmptyTree ( Node 12 EmptyTree ( Node 20 EmptyTree
EmptyTree )))) EmptyTree
```

Well! What about Either a b ? Can it be a functor? The Functor type class wants type constructors that take a single type parameter but Either takes two. Mmm ... I know! We will partially apply Either by supplying a single parameter so that it only has one free parameter. Here's how the type Either a is a functor in the standard libraries.

```
instance Functor ( Either a ) where
    fmap f ( Right x ) = Right ( f x )
    fmap f ( Left x ) = Left x
```

Well, well, what have we done here? You can see how we have instantiated Either a instead of just Either . This is because ` Either a is a type constructor that takes one parameter, while Either takes two. If fmap were specifically for Either a then its type declaration would be (b -> c) -> Either a b -> Either a c since it is the same as b -> c) -> (Either a) b -> (Either a) c . In the implementation, we mapped in the case of the Right type constructor , but we did not map it in the case of Left . Why? Well, if we go back to see how the Either type is defined to b , we vary something like:

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

Well, if we wanted to map a function to both, a and b should have the same

type. I mean, if we wanted to map a function that takes a string and returns another string and b is a string but a is a number, this would not work. Also, looking at fmap if it only operated with Either values , we would see that the first parameter has to remain the same while the second can vary and the first parameter is associated with the Left data constructor .

This also fits in with our box analogy if we think of Left as a kind of empty box with an error message written on one side telling us why the box is empty.

Data.Map dictionaries are also functors as they may (or may not) contain values. In the case of Map k v , fmap would map a function v -> v ' on a Map k v dictionary and return a dictionary of the type Map k v' .

Note

Notice that 'it doesn't have any special meaning in the types in the same way that they don't have any special meaning when naming values. It is usually used to refer to things that are similar, only a little changed.

Try to imagine how the instance of Map k for Functor is created yourself!

With the Functor type class we have seen how type classes can represent interesting higher order concepts. We've also had a bit of practice partially applying types and creating instances. In one of the following chapters we will look at some of the laws that apply to functors.

Note

Functors must obey some laws so that they have properties that we can depend on so we don't have to think much later. If we use fmap (+1) on a list [1,2,3,4] we hope to obtain [2,3,4,5] and not its inverse, [5,4,3,2] . If we use fmap (\ a -> a) (the identity function, which simply returns its parameter) on a list, we hope to get the same list as a result. For example, if we give a wrong instance to our Tree type , when using fmap in a tree where the left sub-tree of a node only contains elements smaller than the node and the right sub-tree only contains elements greater than the node could. produce a tree where this is not true. We will look at the laws of functors in more detail in a future chapter.

Families and martial arts

Type constructors take other types as parameters and end up producing concrete types. This reminds me of functions, which take values as parameters and produce values. We have seen that type constructors can be partially applied (Either String is a type constructor that takes a type and returns a specific type, like Either String Int), just like functions. Very interesting. In this section, we will formally define how types are applied to type constructors, in this section we will formally define how values are applied to functions using type declarations. **You don't need to read this section to continue your search for wisdom about Haskell** and you can't understand it, don't worry. However, doing so will give you a deep understanding of the type system.

So, values like 3 , "YEAH" or takeWhile (the functions are also values since we can use them as parameters) have their corresponding types. Types are a small label that carry values in a way that allows us to reason about them. But the guys have their own little labels, called **families** . A family is more or less the type of a type. It may sound a bit convoluted and confusing, but it is actually a very interesting concept.

What are families and what are they useful for? Well, let's examine the family of a type using the command : k on GHCi.

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

A star? Intriguing ... what does it mean? A * means that the type is a specific type. A specific type is a type that does not take any type parameters and values can only have types that are concrete types. If I had to read * out loud (so far I haven't had to), I'd say *star* or just *guy* .

Okay, now let's see what the Maybe family is .

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

The Maybe type constructor takes a specific type (like Int) and then returns a specific type like Maybe Int . And this is what the family is telling us. In the same way that Int -> Int represents a function that takes an Int and returns an

Int , * -> * represents a type constructor that takes a specific type and returns another specific type. Let's apply the type parameter to Maybe and see what its family is.

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

Just as I expected! We have passed a type parameter to Maybe and we have obtained a specific type (this is what * -> * means). A simile (although not equivalent, types and families are two different things) would be if we did : t isUpper and : t isUpper 'A' . isUpper has the type Char -> Bool and isUpper 'A' has the type Bool since its value is basically True .

We use : k with a type to obtain its family, in the same way that we use : t with a value to obtain its type. As we have already said, the types are the labels of the values and the families are the labels of the types and there are similarities between the two.

Let's go see another family.

```
ghci > : k Either
Either :: * -> * -> *
```

Aha! This tells us that Either takes two specific types as type parameters and produces a specific type. It also looks like a type declaration of a function that takes two values and returns something. Type constructors are curred (like functions), so we can partially apply them.

```
ghci > : k Either String
Either String :: * -> *
ghci > : k Either String Int
Either String Int :: *
```

When we wanted Either to be part of the Functor type class , we had to partially apply it since Functor wants types that take a single parameter`, `while Either takes two. In other words, Functor wants family types * -> * and so we had to partially apply Either to get a family * -> * instead of its original family * -> * -> * . If we look at the definition of Functor again

```
class Functor f where
    fmap :: ( a -> b ) -> f a -> f b
```

We will see that the variable of type f is used as a type that takes a type and produces a specific type. We know that it produces a specific type because it is used as the type of a value in a function. We can deduce that the guys who want Functor friendly must be from the * -> * family .

Now we are going to practice some martial arts. Take a look at the class of types I'm going to use:

```
class Tofu t where
    tofu :: j a -> t a j
```

It seems complicated. How could we create a type that had an instance for this strange type class? Well, let's see what family has to have. Since j a is used as the type of the value that the tofu function takes as a parameter, j a must have the family * . We assume * for a so that we can infer that j belongs to the family * -> * . We see that t also has to produce a specific type and it takes two types. Knowing that a is from the family * and j from * -> * , we can infer that t is from the family * -> (* -> *) -> * . So it takes a concrete type (a), a type constructor (j) that takes a concrete type and returns a concrete type. Wau.

Okay, let's create a type with a family * -> (* -> *) -> * . Here is a possible solution.

```
data Frank a b = Frank { frankField :: b a } deriving ( Show )
```

How do we know that this type belongs to the family * -> (* -> *) -> * ? Well, the fields of a TDA (algebraic data types, *ADT*) serve to contain values, so they obviously belong to the family * . We assume * for a , which means that b takes a type parameter and therefore belongs to the * -> * family . Now that we know the families of a and b since they are parameters of Frank , we see that Frank belongs to the family * -> (* -> *) -> * . The first * represents a and (* -> *) represents b . We are going to create some Frank values and check their types.

```
ghci > : t Frank { frankField = Just "HAHA" }
Frank { frankField = Just "HAHA" } :: Frank [ Char ] Maybe
ghci > : t Frank { frankField = Node 'a' EmptyTree EmptyTree }
Frank { frankField = Node 'a' EmptyTree EmptyTree } :: Frank Char Tree
ghci > : t Frank { frankField = "YES" }
Frank { frankField = "YES" } :: Frank Char []
```

Hmm ... Since frankField has the type in the form of a b , its values must have types of similar form. It can be like Just "HAHA" , which has the type Maybe [Char] or it can be like ['Y', 'E', 'S'] that has the type [Char] (if we used our type of lists that we created previously, it would be List Char). And we see that the types of Frank's values correspond to the Frank family . [Char] belongs to the family * and Maybe belongs to * -> * . Since in order to have values, a type must be a specific type and therefore must be completely applied, each value of Frank bla blaaa belongs to the family * .

Creating the Frank instance for Tofu is pretty simple. We have seen that tofu takes a j a (which for example could be Maybe Int) and returns a t j a . So if we replace j by Frank , the type of the result would be Frank Int Maybe .

```
instance Tofu Frank where
    tofu x = Frank x
ghci > tofu ( Just 'a' ) :: Frank Char Maybe
Frank { frankField = Just 'a' }
ghci > tofu [ "HELLO" ] :: Frank [ Char ] []
Frank { frankField = [ "HELLO" ]}
```

It is not very useful, but we have warmed up. We are going to continue doing martial arts. We have this type:

```
data Barry t k p = Barry { yabba :: p , dabba :: t k }
```

And now we want to create an instance for the Functor class . Functor requires types whose family is * -> * but Barry does not appear to belong to that family. What is Barry's family ? Well, we see that it takes three type parameters, so it will be something like something -> something -> something -> * . It is clear that p is a specific type and therefore belongs to the family * . For k we assume * y by extension, t belongs to * -> * . Now we just have to replace these family by *algos* that we used and see that the type belongs to the family (* -> *) -> * ` ` -> * -> * . Let's check it with GHCi.

```
ghci > : k Barry
Barry :: ( * -> * ) -> * -> * -> *
```

Ah, we were right. Now, to make this type part of the Functor class we have to partially apply the first two type parameters so that we are left with * -> * . This means that we will start with our instance declaration like this: instance

Functor (Barry a b) where . If we see fmap as if made exclusively para` Barry , a guy would fmap :: (a -> b) -> Barry c d a -> Barry c d b because they simply have replaced the f of Functor by Barry c d . The third type parameter of Barry would have to change and in this way we would have:

```
instance Functor ( Barry a b ) where
    fmap f ( Barry { yabba = x , dabba = y }) = Barry { yabba = f x , dabba = y }
```

There you have it! We have simply applied f to the first field.

In this section, we have taken a good look at how type parameters work and how they are formalized with families, in the same way that we formalize function parameters with type declarations. We have seen that there are similarities between functions and type constructors. Anyway, they are totally different things. When we work with Haskell, you usually don't have to worry about families or mentally infer families like we've done here. Normally, you have to partially apply your type to * -> * or * when creating an instance for some class from the standard library, but it's nice to know how it really works. It is interesting to know that guys have their own little guys too. Again, you don't have to understand everything we've just done here, but if you understand how families work, you have a better chance of understanding Haskell's type system correctly.

Chapter VII

Input and output

We have already mentioned that Haskell is a purely functional programming language. Whereas in an imperative language you usually get results by giving the computer a series of steps to execute, functional programming is more like defining what things are. In Haskell, a function cannot change a state, like changing the content of a variable (when a function changes state, we say it has *side effects*). The only thing a Haskell function can do is return some result to us based on the parameters we give it. If a function is called twice with the same parameters, it has to return the same result. While this may seem a bit restrictive from the point of view of an imperative world, we have already seen how it is actually a great thing. In imperative language, you have no guarantee that a function that should only play with a few numbers will not burn your house, kidnap your dog, or grate your car with a potato while playing with those numbers. For example, when we do a binary search with a tree, we don't insert any element in the tree modifying any node. Our function to insert an element into a tree actually returns a new tree, since it cannot modify the previous tree.

As the fact that the functions are not capable of changing the state is a good thing, as it helps us to reason about our programs, there is a problem with this. If a function cannot change anything in the world, how is it supposed to tell us the result it has calculated? To get him to tell us what he's calculated, he has to change the state of an output device (usually the state of the screen), which will emit photons that travel through our brains to change the state of our mind, awesome.

Do not despair, all is not lost. Haskell actually has a very clever system for dealing with functions that have side effects in a way that separates the part of our program that is pure from the part of our program that is impure, which does all the dirty work of keyboard speaking. and the screen. With these parts well separated, we can continue to reason about our pure program and take advantage of all that purity offers us, such as lazy evaluation, security, and modularity as we communicate with the outside world.

Hello World!

Until now, we've always loaded our features into GHCi to test and play with. We have also explored the functions of the standard library in this way. But now, after eight chapters, we're finally going to write our first real Haskell show, Wau! And of course, we are going to create the mythical `"hello world!"` .

Note

For the purposes of this chapter, I am going to assume that you are using a *unix* system to learn Haskell. If you are on *Windows* , I suggest you download cygwin , which is a *Linux* environment for *Windows* , or in other words, just what you need.

So to get started, paste the following into your favorite text editor:

```
main = putStrLn "hello, world"
```

We have just defined the `main` function and inside it we call a `putStrLn` function with the parameter `"hello, world"` . It seems ordinary, but it is not, as we will see in a moment. Save the file as `helloworld.hs` .

And now we are going to do something we have never done before. Let's compile a program! Aren't you nervous Open a terminal and navigate to the directory where `helloworld.hs` is located and do the following:

```
$ ghc --make helloworld
[1 of 1] Compiling Main (helloworld.hs, helloworld.o)
Linking helloworld ...
```

Voucher! With a little luck you will have obtained something similar and you will be able to run the program by doing `./helloworld` .

```
$ ./helloworld
hello world
```

There you have it, our first compiled program that displays a message from the terminal. Extraordinarily boring!

Let's examine what we have written. First, let's look at the type of the `putStrLn` function .

```
ghci>: t putStrLn
putStrLn :: String -> IO ()
ghci>: t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

We can read the type of putStrLn as: putStrLn takes a string and returns an IO action that returns a type () (i.e. the empty tuple, also known as a unit). An IO action is something that when performed will load an action with some side effect (such as reading from input or displaying things on screen) and will contain some kind of value within it. Showing something on screen doesn't really have any result value, so the dummy value () is used .

Note

The empty tuple has the value of () and also has the type () . Something like data Nothing = Nothing .

And when is an IO action executed ? Well, this is where main comes in . An IO action is executed when we give it the name ` main and run our program.

Having your entire program in one IO action may seem a bit restricted. For this reason we can use the syntax do to join several IO actions in one. Take a look at the following example:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey" ++ name ++ ", you rock!")
```

Ah ... interesting New syntax! It is read in a similar way to that of an imperative program. If you compile it and run it, it will probably behave as you expect. Notice that we have used a do and then we have put in a series of steps, just like in an imperative program. Each of these steps is an IO action . We putting all of them together in the same block do get one action IO . The action we get has the type IO () because that is the type of the last action within the block.

For this reason, main always has to have the IO type something , where something is some concrete type. By convention, the type declaration of main is not usually specified .

An interesting thing we haven't seen before is on the third line, which is name <- getLine . It seems as if you were reading a line of text and saving it to a variable called name , really? Well, let's examine the type of getLine .

```
ghci>: t getLine
getLine :: IO String
```

Voucher. `getLine` is an `IO` action that contains a result of type `String` . It seems to make sense since it will wait for the user to type something into the terminal and then that something will be represented with a string. So what about `name <- getLine` ? You can read that piece of code like: perform the `getLine` action and then `bind` the result to the `name` value . `getLine` has the `IO String` type , so `name` will have the `String` type . You can imagine an `IO` action as a box with legs that will go out into the real world and do something there (like paint a graffiti on a wall) and maybe come back with some data inside. Once that data has been brought in, the only way to open the box and take the data from inside is to use the `<-` construct . And if we are extracting data from an `IO` action , we can only extract it when we are inside some `IO` action . This is how Haskell manages and separates the pure and impure parts of our code. In that sense `getLine` is impure since the result is not guaranteed to be the same when called twice. This is why your result is *contaminated* with `IO` type constructor and we can only extract this data within an `IO` code . And since the `IO` code is also contaminated, each calculation that depends on an `IO`-contaminated data will have a contaminated result as well.

When we say *contaminated* , we don't mean that we will never again be able to use the result contained in an `IO` action in our pure code. No, when we bind a value contained in an `IO` action to a name, we temporarily *decontaminate it* . When we make `name <- getLine` , `name` is a normal string, since it represents what is inside the box. We can have a really complicated function that, let's say, takes your name (a normal string) as a parameter and predicts your luck and your entire future based solely on your name. It could be something like this:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future:" ++ tellFortune name
```

`tellFortune` (or any other function passed `name`) doesn't have to know anything about `IO` , it's just a normal function of type `String -> String` .

Look at this piece of code. Is it valid?

```
nameTag = "Hello, my name is" ++ getLine
```

If you said no, you can go for a cookie. If you said yes, you see forgetting whims. The reason this doesn't work is that `++` requires that its two parameters be of the same list type. The parameter on the left has the `String`

type (or `[Char]` if you prefer), while the `getLine` has the `IO String` type . We cannot concatenate a string with an `IO` action . We must first extract the result of the `IO` action to get a value of type `String` and the only way to get it is to do something like `name <- getLine` inside an `IO` action . If we want to deal with impure data we have to do it in an impure environment. The impurity stain spreads like a plague through our code and it is our duty to keep the `IO` parts as small as possible.

Every `IO` action that is executed has a result encapsulated with it. For this reason we could have written the above code as:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey" ++ name ++ ", you rock!")
```

However, `foo` would simply have the value `()` which is not very useful. Note that we have not linked the last `putStrLn` to any name. This is because in a `do` block , **the last action cannot be linked** like the first two. When we venture into the world of monads we will see the specific reason for this restriction. For now, you may think that a `do` block automatically extracts the value of the last action and links it to its own result.

Except for the last line, each line in a `do` block that is not bound can also be written as a ligature. So `putStrLn "Blah"` can be written as `_ <- putStrLn "Blah"` . However it is useless, so we don't use `<-` for actions that don't contain an important result, like `putStrLn something` .

Beginners sometimes think that doing things like `name = getLine` will read a line through the entry and bind it to `name` . Well, no, what this does is give the `getLine` action a new name, called `name` . Remember that to get the value contained within an `IO` action , you have to bind it to a name with `<-` inside another `IO` action .

`IO` actions are only executed when given the name of `main` or when they are inside a larger `IO` action that we have composed with a `do` block . We can use a `do` block to put some `IO` actions together and then use that `IO` action inside another `do` block and so on. Either way, in the end they will only run when reached by `main` .

Oh right, there is also another case where `IO` actions are executed. When we write an `IO` action in GHCi and press enter.

```
ghci> putStrLn "HEEY"
HEEY
```

Even when we write a number or a function call in GHCi, it will evaluate it (as much as you need) and then call `show` to show that string in terminal using `putStrLn` implicitly.

Do you remember the `let` sections ? If not, refresh your memory by reading this *section* . They have the form `let ligatures in expression` , where `ligatures` are the names given to the expressions and `expression` will be the expression where they will be evaluated. We also said that comprehension lists did not need the `in` part . Well, you can use them in a `C` block practically the same as for comprehension lists. Watch this:

```
mport Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey" ++ bigFirstName ++ "" ++ bigLastName ++ ", how are you?"
```

Do you see how the `IO` actions inside the `do` block are aligned? Also note how the `let` section is aligned with `IO` actions and the `let` names are aligned with each other. It is good practice to do this, since bleeding is important in Haskell. We have made `map toUpper firstName` , which converts something like `"john"` to the string `"JOHN"` . We have bound that uppercase string to a name and then used it in a string to display it by the terminal.

You may be wondering when to use `<-` and when to use `let` . Well, remember that `<-` is (for now) to execute `IO` actions and link their results. However, `map toUpper firstName` is not an `IO` action . It is a pure expression of Haskell. So we use `<-` when we want to link the results of an `IO` action while we use `let` to link pure expressions. If we had done something like `let firstName = getLine` , we would simply have given the `getLine` action a new name and would still need to use `<-` to execute the action.

Now we are going to create a program that continuously reads a line and displays that line with its words backwards. The execution of the program

will stop when it finds an empty line. Here is the program.

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords. map reverse. Words
```

To understand how it works, you can run the program before reading the code.

Note

To run a program you can either compile it by producing an executable and then run it using `ghc --make helloworld` and then `./helloworld` or you can use the `runhaskell` command like this: `runhaskell helloworld.hs` and your program will be run on the fly.

Let's take a look at the `reverseWords` function first. It is just a normal function that takes a string like `"hey there man"` and then calls `words` which produces a list of words like `["hey", "there", "man"]`. Then we map `reverse` over the list, getting `["yeh", "ereht", "nam"]`, then we have a single string again using `unwords` and the end result is `"yeh ereht nam"`. Notice how we have used the composition of functions. Without the composition of functions we should have written something like `reverseWords st = unwords (map reverse (words st))`.

What about `main`? First, we get a line from the terminal by running `getLine` and call it `line`. And now we have a conditional expression. Remember that in Haskell, each `if` must have its `else` since every expression must have some kind of value. We use the condition so that when it is true (in our case, when the line is empty) we perform an `IO` action and when it is not, we perform the action located in the `else`. For this reason the conditions inside an `IO` action have the form `if condition then action else action`.

Let's have a look at what happens under the `else` clause. Since we must have exactly one `IO` action after the `else` we have to use a `do` block to put all the

actions together. It could also be written like this:

```
else (do
    putStrLn $ reverseWords line
    main)
```

This makes the fact that a `do` block is seen as a single `IO` action more explicit , but it's uglier. Either way, inside the `do` block we call `reverseWords` on the line we got from `getLine` and then display the result by the terminal. After this, we simply run `main` . It is called recursively and there is no problem since `main` is itself an `IO` action . In a way it is as if we were going back to the beginning of the program.

Now what happens when `null line` evaluates to true? The action after the `then` is executed . If we search we will see that it puts `then` return ()` . If you know of any imperative language like *C* , *Java Python* , you're probably thinking that you already know what `return` is and that you can skip this long paragraph. Well, **the** `return` **of Haskell has nothing to do with the** `return` **of most other languages** . It has the same name, which confuses many people, but it is actually very different. In imperative languages, `return` usually ends the execution of a method or a subroutine and returns some kind of value to whoever called it. In Haskell (within `IO` actions specifically), what it does is convert a pure value into an `IO` action . If you think of it like the box analogy we saw, `return` takes a value and puts it inside a box. The resulting `IO` action really does nothing, it simply has that value as a result. So in an `IO` context , `return "haha"` will have the type `IO String` . What is the reason for transforming a pure value into an action that really does nothing? Why contaminate our program more with `IO` ? Well we need some `IO` action in case we find an empty line. For this reason we have created an `IO` action that really does nothing with `return ()` .

By using `return` we do not cause a `do` block to end its execution or anything like that. For example, this program will run to the last line without any problem.

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
```

```
    putStrLn line
```

All these returns do is create IO actions that actually do nothing except contain a value, which is wasted since it is not bound to any name. We can use return in combination with <- to bind things to names.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

As you can see, return is in a way the opposite of <- . While return takes values and puts them in a box, <- takes a box (and executes it) and outputs the value it contains, binding it to a name. However doing these things is a bit redundant since you can use let sections to achieve the same:

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

When dealing with blocks do IO , we normally use return or because we want to create an action IO do nothing or because we want the result to host the action IO resulting from a block do not the value of the last action.

Note

A do block can contain a single IO action . In that case, it is the same as writing just that action. There are people who prefer to write then do return () in this case since the else also has a do .

Before we see how to deal with files, let's have a look at some functions that are useful when working with IO .

- putStr is very similar to putStrLn in that it takes a string and returns an action that will print that string through the terminal, only that putStr does not jump to a new line after printing the string as putStrLn does.

```
main = do putStr "Hey,"
          putStr "I'm"
          putStrLn "Andy!"
$ runhaskell putstr_test.hs
```

- Hey, I'm Andy!

Its type is `putStr :: String -> IO ()` , so the result contained in the `IO` action is unity. A useless value, so there is no point in linking it to anything.

- `putChar` takes a character and returns an `IO` action that will be printed by the terminal.

- main = do putChar 't'
- putChar 'e'
- putChar 'h'
- $ runhaskell putchar_test.hs
- teh

`putStr` is actually defined recursively with the help of `putChar` . The base case is the empty string, so if we are printing the empty string, we simply return an `IO` action that does nothing using `return ()` . If it is not empty, we print the first character of the string using `putChar` and then print the rest of the string using `putStr` .

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x: xs) = do
    putChar x
    putStr xs
```

Note that we can use recursion in `IO` in the same way that we do in pure code. Just like in pure code, we define the base case and then think that it is really the result. It is an action that first prints the first character and then prints the rest of the string.

- `print` takes a value of any type that is a member of the `Show` class (so we know it can be represented as a string), calls `show` with that value to get its representation, and then displays that string by terminal. Basically it is `putStrLn. show` . First run `show` with a value and then feed `putStrLn` with that value, which returns an action that will print our value.

- main = do print True
- print 2
- print "haha"
- print 3.2
- print [3,4,3]
- $ runhaskell print_test.hs
- True
- two

- "Haha"
- 3.2
- [3,4,3]

As you can see, it is a very useful function. Remember when we talk about `IO` actions executing only when they are reached by `main` or when we try to evaluate them in GHCi? When we type a value (such as `3` or `[1,2,3]`) and press enter, GHCi actually uses `print` with that value to display it by the terminal.

```
ghci> 3
3
ghci> print 3
3
ghci> map (++ "!") ["hey", "ho", "woo"]
["hey!", "ho!", "woo!"]
ghci> print (map (++ "!") ["hey", "ho", "woo"])
["hey!", "ho!", "woo!"]
```

When we want to print strings we usually use `putStrLn` since we usually want the double quotes surrounding the representation of a string, but `print` is usually used to display values of any other type .

- `getChar` is an `IO` action that reads a character through standard input (keyboard). Therefore, its type is `getChar :: IO Char` , since the result contained within the `IO` action is a character. Note that due to *buffering* , the action of reading a character is not executed until the user presses the enter key.

```
main = do
   c <- getChar
   if c / = ''
      then do
         putChar c
         main
      else return ()
```

This program seems to read a character and check if it is a space. If it is, it stops the program execution and if it is not, it prints it by the terminal and then repeats its execution. Well, it looks like it does this, but it doesn't do it the way we expect. Check it.

```
$ runhaskell getchar_test.hs
hello sir
Hello
```

The second line is the exit. We typed `hello sir` and then we hit

enter. Due to *buffering* , program execution only starts after executing intro and not after each character pressed. Once we press enter, it acts as if we had written those characters from the beginning. Try to play around with this program to understand how it works.

- The `when` function is in the `Control.Monad` module (to access it, `import Control.Monad`). It's interesting since inside a `do` block it looks like it's a flow control statement, but it's actually a normal function. It takes a boolean value and an `IO` action so that if the boolean value is `True` , it will return the same action that we supply it. However, if it is false, it will return an action `return ()` , an action that does absolutely nothing. Here is how we could have written the previous piece of code that showed the use of `getChar` using `when` :

```
import Control.Monad

main = do
   c <- getChar
   when (c / = '') $ do
      putChar c
      main
```

As you can see, it is useful to encapsulate the *`if* pattern *something then do the else return () `action* . There is also the `unless` function that is exactly the same as `when` only that returns the original action when `False is` found instead of `True` .

- `sequence` takes a list of `IO` actions and returns an action that will perform all those actions one after the other. The result contained in the `IO` action will be a list with all the results of all the `IO` actions that were executed. Its type is `sequence :: [IO a] -> `IO [a]` . Do this:

```
main = do
   a <- getLine
   b <- getLine
   c <- getLine
   print [a, b, c]
```

It is exactly the same as doing:

```
main = do
    rs <- sequence [getLine, getLine, getLine]
```

```
print rs
```

So `sequence [getLine, getLine, getLine]` creates an `IO` action that will execute `getLine` three times. If we link that action to a name, the result will be a list that will contain all the results, in our case, a list with three lines that the user has entered.

A common use of `sequence` is when we map functions like `print` or `putStrLn` on lists. Doing `map print [1,2,3,4]` does not create an `IO` action . It will create a list of `IO` actions , since it is the same as if we wrote `[print 1, print 2, print 3, print 4]` . If we want to transform that list of actions into a single IO action, we have to sequence it.

```
ghci> sequence (map print [1,2,3,4,5])
one
two
3
4
5
[(), (), (), (), ()]
```

What's that about `[(), (), (), (), ()]` ? Well, when we evaluate an `IO` action in GHCi it is executed and its result is shown on the screen, unless the result is `()` , in which case it is not shown. For this reason when evaluating `putStrLn "hehe"` GHCi only prints `"hehe"` (since the result contained in the `putStrLn "hehe" action` is `()`). However when we use `getLine` in GHCi, the result of that action is printed on screen, since `getLine` has the `IO String` type .

- Since mapping a function that returns an `IO` action on a list and then sequencing it is very common, the auxiliary functions `mapM` and `mapM_` were introduced . `mapM` takes a function and a list, maps the function over the list, and then the sequence. `mapM_` does the same thing, only then gets rid of the result. We usually use `mapM_` when we don't care about the result of the sequenced actions.

```
ghci> mapM print [1,2,3]
one
two
3
[(), (), ()]
ghci> mapM_ print [1,2,3]
one
two
3
```

- `forever` takes an `IO` action and returns another `IO` action that will simply repeat the first action indefinitely. It is located in `Control.Monad` . This little program will ask the user for a string and then return it in uppercase, indefinitely:

```
import Control.Monad
import Data.Char

main = forever $ do
   putStr "Give me some input:"
   l <- getLine
   putStrLn $ map toUpper l
```

- `forM` (located in `Control.Monad`) is like `mapM` only it has its parameters changed from site. The first parameter is the list and the second is the function to map over the list, which will then be sequenced. What is it useful for? Well, with creative use of lambda functions and `do` notation we can do things like these:

```
import Control.Monad

main = do
   colors <- forM [1,2,3,4] (\ a -> do
      putStrLn $ "Which color do you associate with the number" ++ show a ++ "?"
      color <- getLine
      return color)
   putStrLn "The colors that you associate with 1, 2, 3 and 4 are:"
   mapM putStrLn colors
```

`(\ a -> do ...)` is a function that takes a number and returns an `IO` action . We have to surround it with parentheses, otherwise the lambda function would think that the last two lines belong to it. Notice that we use `return color` inside the `do` block . We do it this way so that the `IO` action that defines the `do` block results in the color we want. We don't really have to do them because `getLine` already has it contained. Doing `color <- getLine` and then doing `color return` is simply extracting the result from `getLine` and then inserting it again, so it's the same as doing just `getLine.` `` ` ForM`` (called with its two parameters) produces an action `IO` , the result of which we will link to `colors` . `colors` is a simple list containing strings. In the end, we print all those colors doing `mapM putStrLn colors` .

You can see it in the sense that `forM` creates an `IO` action for each item in a list. What each action does will depend on the element that has been used to create the action. In the end, perform all those actions and link all the results to something. We don't have to bind it, we can just throw it away.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
net
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
net
orange
```

We could actually have done the same without `forM` , only with `forM it` is more readable. Normally we use `forM` when we want to map and sequence some actions that we have defined using the notation `do` . Similarly, we could have replaced the last line with `forM colors putStrLn` .

In this section we have learned the basics of entry and exit. We have also seen what `IO` actions are , how they allow us to perform input and output actions and when they are actually executed. `IO` shares are securities just like any other value in Haskell. We can pass them as parameters in functions, and functions can return actions as results. What is special is when they are reached by `main` (or are the result of a GHCi sentence), they are executed. And that's when they can write things on your screen or play through your speakers. Each `IO` action can also contain a result that will tell us what it has been able to obtain from the real world.

Don't think of the `putStrLn` function as a function that takes a string and prints it on the screen. Think of it as a function that takes a string and returns an `IO` action . That `IO` action , when executed, will print that string on the screen.

Files and data streams

getChar is an I / O action that reads a single character from the terminal. getLine is an I / O action that reads a line from the terminal. These functions are fairly simple, and most languages have similar functions or statements. But now we are going to see getContents . getContents is an I / O action that reads anything from standard input until it finds an end-of-file character. Its type is getContents :: IO String . The good thing about getContents is that it performs lazy I / O. When we do foo <- getContents , it doesn't read all the input data at once, it stores it in memory and then binds it to foo . No, he's lazy! It will say "Yes, yes, I'll read the terminal entry later, when you really need it."

getContents is really useful when we are redirecting the output of one program to the input of another program. In case you don't know how redirection works on *unix* systems , here is a short introduction. We are going to create a text file containing this little :

```
I'm a lil 'teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Yes, you are right, this haiku sucks. If you know any good haikus guide let me know.

Now remember that little program we wrote when we explained the forever function . It would ask the user for a line and return it in capitals, then go back to doing the same thing indefinitely. Just so you don't have to scroll back, here's the code again:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input:"
    l <- getLine
    putStrLn $ map toUpper l
```

We are going to save this program as capslocker.hs or something similar and compile it. And now, we are going to use *unix* redirects to supply our text file directly to our little program. We are going to help ourselves by using the GNU cat program , which shows the content of the file that we pass to it as a parameter through the terminal. Look!

```
$ ghc --make capslocker
[1 of 1] Compiling Main (capslocker.hs, capslocker.o)
```

```
Linking capslocker ...
$ cat haiku.txt
I'm a lil 'teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL 'TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

As you can see, to redirect the output of one program (in our case `cat`) to the input of another (`capslocker`) it is achieved with the character `|`. What we just did would be equivalent to running `capslocker`, typing our haiku in the terminal, and then entering the end of file character (usually this is done by pressing *Ctrl + D*). It is like running `cat haiku.txt` and saying: "Wait, don't show this on the screen, pass it to `capslocker`".

So what we're doing using `forever` is basically taking the input and transforming it into some kind of output. For this reason we can use `getContents` to make our program better and even shorter.

```
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

We execute the `getContents` I / O `action` and name the string it produces as `contents`. Then we `trace toUpper` over the string and display the result by terminal. Note that the strings are basically ready, which are lazy, and `getContents` is a lazy I / O action. Therefore it will not try to read all the content at once to save it in memory before displaying it in capital letters by the terminal. It will actually show the uppercase version as you read as it only reads one line of the entry when you really need it.

```
$ cat haiku.txt | ./capslocker
I'M A LIL 'TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

Great, it works. What if we run `capslocker` and try to write lines of text ourselves?

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

Out by pressing *Ctrl + D* . As you can see, it shows our input in capital letters line by line. When the result of `getContents` is bound to `contents` , it is not represented in memory as a real string, but rather as a promise that it will eventually produce a string. When we `plot toUpper` over `contents` , there is also a promise that that function will plot over the final content. Finally, when `putStr` is run it says to the promise above: "Hey! I need an uppercase line!". That's when `getContents` actually reads the input and passes a line to the code that has asked it to produce something tangible. That code traces `toUpper` on that line and passes the result to `putStr` , and it takes care of displaying it. Then `putStr` says, "Hey, I need the next line. Come on!" and it is repeated until there is no more data in the entry, which is represented by the end of file character.

We are going to create a program that takes a few lines and then only displays those that are less than 10 characters long. Observe:

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\ line -> length line <10) allLines
        result = unlines shortLines
    in result
```

We have made the part of our program dedicated to I / O as small as possible. Since our program is supposed to take an input and display an output based on the input, we can implement it by reading the contents of the input, executing a function on them, and then showing what that function returns to us.

The `shortLinesOnly` function works like this: it takes a string, like `"short \ nlooooooooooooooong \ nshort again"` . This chain has three lines, two of them are short and the middle one is long. Run the `lines` function on that string, so we get `["short", "looooooooooooooong", `` "short again"]` which we then `link` to `allLines` . This list of strings is then filtered so that only lines that are less than 10

characters in length remain in the list, producing `["short", "short again"]` . Finally `unlines` concatenates the list into a single string, returning `"short \ nshort again"` . We're going to try it.

```
i'm short
so am i
i am a looooooooooong line !!!
yeah i'm long so what hahahaha !!!!!!
short line
looooooooooooooooooooooooooong
shorts
$ ghc --make shortlinesonly
[1 of 1] Compiling Main (shortlinesonly.hs, shortlinesonly.o)
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
shorts
```

We redirect the contents of `shortlines.txt` to the input of `shortlinesonly` , so that we only get short lines.

This pattern of taking a string as input, transforming it with a function, and displaying the result of that transformation is so common that there is a function that makes this easier, the `interact` function . `interact` takes a function of type `String -> String` as a parameter and returns an I / O action that will take the input of the program, execute the function on it and display the result of this function on the screen. We are going to modify our program to use this function.

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\ line -> length line <10) allLines
        result = unlines shortLines
    in result
```

In order to show that we can achieve the same thing with much less code (even if it is a little less readable) and to demonstrate our function composition skills, we are going to modify it a little more.

```
main = interact $ unlines. filter ((<10). length). lines
```

Wau We've reduced it to a single line of code!

`interact` can be used to create programs to which some content will be redirected and then display a result, or to create programs that appear to read

a line written by the user from the input, display a result based on that line, and then continue with another line. There is actually no difference between them, it just depends on how the user uses it.

We are going to create a program that continuously reads a line and tells us if that line is a palindrome or not. We could simply use `getLine` to read a line, show the user whether it is palindrome or not, and rerun `main` . But it is simpler if we use `interact` . When you use `interact` , think about what you have to do to transform the program's input into the output you want. In our case, we have to replace each line of the entry in `"palindrome"` or `"not a palindrome"` . So we have to transform something like `"elephant \ nABCBA \ nwhatever"` into `"not a palindrome \ npalindrome \ nnot a palindrome"` Let's try it!

```
answer Palindromes contents = unlines (map f (lines contents))
    where is Palindrome xs = xs == reverse xs
          f xs = if is Palindrome xs then "palindrome" else "not a palindrome"
```

Let's write it in dot-free style:

```
answer Palindromes = unlines. map f. lines
    where is Palindrome xs = xs == reverse xs
          f xs = if is Palindrome xs then "palindrome" else "not a palindrome"
```

Simple. First convert something like `"elephant \ nABCBA \ nwhatever"` into `["elephant", "ABCBA", "whatever"]` and then trace `f` over the list, returning `["not a palindrome", "palindrome", "not a palindrome" ]` . Finally use `unlines` to concatenate the list of strings into a single string. Now we can do:

```
main = interact respondPalindromes
```

Let's check it out.

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

Even though we have created a program that transforms one large input string into another, it acts as if we have made a program that reads line by line. This is because Haskell is lazy and wants to display the first line of the result, but he cannot do it because he doesn't have the first line of the input yet. So as soon as you have the first line of the input, it will show the first line of the

output. We exit the program using the end of file character.

We can also use the program redirecting the content of a file. Let's say we have this file:

```
dogaroo
Radar
rotor
madam
```

And we have saved it as `words.txt` . This would be how we would redirect the file to the entry of our program.

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

Again, we get the same output as if we had run our program and typed in the words ourselves. We just don't see the `palindromes.hs` entry because it has been redirected from a file.

You probably already know how lazy I / O works and how you can take advantage of it. You can think in terms of how the output is supposed to be and write a function that does the transformation. In lazy I / O, nothing is consumed from the input until it really has to be done, that is, when we want to show something on the screen that depends on the input.

Until now, we have worked with I / O showing and reading things from the terminal. But what about writing and reading files? Well, in a way, we already have. You may think that reading something from the terminal is like reading something from a special file. The same happens when writing to the terminal, it is similar to writing to a file. We can call these two files `stdout` and `stdin` , which represent standard output and standard input respectively. With this in mind, we will see that writing and reading files is very similar to writing to standard output and reading from standard input.

We'll start with a really simple program that opens a file called `girlfriend.txt` , which contains a verse from *Avril Lavigne's* hit # 1 , *Girlfriend* , and displays it from the terminal. Here's `girlfriend.txt` :

```
Hey Hey You! You!
I don't like your girlfriend!
```

```
No way! No way!
I think you need a new one!
```

And here is our program:

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Executing it, we obtain the expected result:

```
$ runhaskell girlfriend.hs
Hey Hey You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

We are going to analyze it line by line. The first line is just four exclamations trying to get our attention. On the second line, Avril tells us that she doesn't like our current partner. The third line aims to emphasize their disagreement, while the fourth suggests that we find a new girlfriend.

Great! Now we are going to analyze our program line by line. The program has several I / O actions attached in a `do` block . In the first line of block `do` we see that there is a new function called `openFile` . Its type is as follows: `openFile :: FilePath -> IOMode -> IO Handle` . If you read it aloud it says: `openFile` takes the path of a file and an `IOMode` and returns an I / O action that will open the indicated file and will contain a manipulator as a result.

```
type FilePath = String
```

`IOMode` is a type that is defined as:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

In the same way that that type that we believe represented the seven days of the week, this type is an enumeration that represents what we want to do with an open file. Very simple. Note that the type is `IOMode` and not `IO Mode` . `IO Mode` would be an I / O action that would contain a value of type `Mode` as a result, but `IOMode` is simply an enumeration.

At the end this function returns an I / O action that will open the indicated file in the indicated way. If we link the action to something at the end we get a `Handle` . A value of type `Handle` represents where our file is. We will use it to manipulate the file so that we know where to read and write data from. It would be a bit stupid to open a file and not bind the manipulator since we couldn't do anything with that file. In our case we bind the manipulator to `handle` .

In the next line we see a function called `hGetContents` . It takes a `Handle` , so it knows where to read the content from, and returns an `IO String` , an I / O action that contains the contents of the file as a result. This function is very similar to `getContents` . The only difference is that `getContents` automatically reads from standard input (i.e. from terminal), while `hGetContents` takes the handle of a file that tells it where to read. Otherwise, they work exactly the same. Like `getContents` , `hGetContents` will not read all the contents of a file at once if it needs it. This is very interesting since we can treat `contents` as if it were all the contents of the file, only it will not actually be loaded in memory. In case we read a huge file, executing `hGetContents` would not saturate the memory since only what is needed will be read.

Notice the difference between the manipulator used to represent the file and the contents of the file, linked in our program to `handle` and `contents` . The manipulator is something that represents the file with which we are working. If you imagine the filesystem as if it were a great book and each file were a chapter of the book, the manipulator would be like a marker that indicates where we are reading (or writing) in a chapter, while the content would be the chapter per se.

With `putStr contents` we simply display the contents of the file by standard output. Then we run `hClose` , which takes a manipulator and returns an I / O action that closes the file. You have to close every file you open with `openFile` yourself !

Another way to do what we just did is to use the `withFile` function , whose type declaration is `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a` . It takes the path of a file, an `IOMode,` and then it takes a function which in turn takes a manipulator and returns an I / O action. `withFile` returns an I / O action that will open the indicated file, do something with it, and then close the file. The result contained in the final I / O action is the same as the result contained in the I / O action of the function passed to it as a parameter. It may sound a bit

complicated to you, but it is really simple, especially with the help of lambdas. Here's our old program rewritten using `withFile` :

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\ handle -> do
        contents <- hGetContents handle
        putStr contents)
```

As you can see, both are very similar. `(\ handle -> ...)` is the function that takes a manipulator and returns an I / O action and this function is usually implemented using lambdas. The reason why you should take a function that returns an I / O action instead of directly taking an I / O action to do something and then close the file, is so that the function we pass you knows which file to operate on. . In this way, `withFile` opens a file and passes the manipulator to the function that we give it. You get an I / O action as a result and then create an I / O action that behaves the same way, only it closes the file first. This is how we would implement the `withFile` function :

```
withFile ':: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile 'path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

We know that the result must be an I / O action so we can start directly with a `do` . First we open the file and obtain the manipulator. Then we apply `handle` to our function and get an I / O action that will do all the work. We bind that action to `result` , close the file and do `return result` . By performing the `return` on the result that contained the I / O action that we got from `f` , we make our I / O action contain the same result that we got from `f handle` . So if `f handle` returns an action that reads a number of lines from the standard input and then writes them to the file, so that it contains the number of lines it has read, the action resulting from `withFile '` will also result the number of lines read.

In the same way that `hGetContents` works the same as `getContents` but on the indicated file, there are also `hGetLine` , `hPutStr` , `hPutStrLn` , `hGetChar` , etc. They work exactly the same as their namesakes, only they take a manipulator as a parameter and operate on the indicated file instead of on the standard input or

output. For example, putStrLn is a function that takes a string and returns an I / O action that will display that string by the terminal followed by a line break. hPutStrLn takes a manipulator and a string and returns an I / O action that will write that string to the indicated file, followed by a line break. Similarly, hGetLine takes a manipulator and returns an I / O action that reads a line from its file.

Loading files and then treating their contents as strings is so common that we have these three small functions that make our lives easier:

- readFile has the type declaration readFile :: FilePath -> IO String . Remember, FilePath is just a synonym for String . readFile takes the path of a file and returns an I / O action that will read that file (lazily) and bind its contents to a string. This is usually more convenient than doing openFile and binding your manipulator and then using hGetContents . Here's what our previous example would look like using readFile :

```
import System.IO

main = do
   contents <- readFile "girlfriend.txt"
   putStr contents
```

Since we do not obtain a manipulator with which to identify our file, we cannot close it manually, so Haskell takes care of closing it for us when we use readFile .

- writeFile has the type FilePath -> String -> IO () . It takes the path of a file and a string to write to that file and returns an I / O action that will take care of writing it. In case the indicated file already exists, it will overwrite the file from the beginning. Here's how to convert girlfriend.txt to a capital version and save it to girlfriendcaps.txt :

```
import System.IO
import Data.Char

main = do
   contents <- readFile "girlfriend.txt"
   writeFile "girlfriendcaps.txt" (map toUpper contents)
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
```

- I DON'T LIKE YOUR GIRLFRIEND!
- NO WAY! NO WAY!
- I THINK YOU NEED A NEW ONE!

- appendFile has the same type as writeFile , only appendFile does not overwrite the file from the beginning in case the indicated file already exists, but instead adds containing to the end of the file.

Let's say we have a file todo.txt that contains a task that we must perform on each line. Now we are going to create a program that takes a line through the standard input and adds it to our task list.

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\ n")
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

We have to add "\ n" to the end of each line since getLine does not return the end of line character at the end.

Oh, one more thing. We have talked about how doing contents <- hGetContents handle does not cause the file in Etero to be read at once and stored in memory. It is lazy I / O action, so by doing this:

```
main = do
    withFile "something.txt" ReadMode (\ handle -> do
        contents <- hGetContents handle
        putStr contents)
```

It is actually like redirecting the file to the output. In the same way that you can treat strings as data streams, you can also treat files as data streams. This will read one line at a time and display it on the screen. You are probably wondering how often the disk is accessed? How big is each transfer? Well, for text files, the default buffer size is one line. This means that the smallest

part of the file that can be read at one time is one line. For this reason the example above actually read one line, displayed it, read another line, displayed it, etc. For binary files, the buffer size is usually one block. This means that binary files are read from block to block. The size of a block is whatever suits your operating system.

You can control exactly how the buffer behaves using the hSetBuffering function . This takes a manipulator and a BufferMode and returns an I / O action that sets the buffer properties for that file. BufferMode is a simple enumeration type and its possible values are: NoBuffering , LineBuffering or BlockBuffering (Maybe Int) . The Maybe Int indicates the size of the block, in bytes. If it is Nothing , the operating system will determine the appropriate size. NoBuffering means that one character will be written or read at a time. Normally NoBuffering is not very efficient since you have to access the disk many times.

Here's our previous example, only this time it will read 2048 byte blocks instead of line by line.

```
main = do
    withFile "something.txt" ReadMode (\ handle -> do
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Reading files with large blocks can help us if we want to minimize disk access or when our file is actually a resource on a very slow network.

We can also use hFlush , which is a function that takes a manipulator and returns an I / O action that will empty the buffer of the file associated with the manipulator. When we use a line buffer, the buffer is flushed after each line. When we use a block buffer, the buffer is flushed after a block is read or written. It is also emptied after closing a manipulator. This means that when we reach a line break, the read (or write) mechanism will report all the data so far. But we can use hFlush to force that data report. After emptying, the data is available to any other program that is running.

To better understand the block buffer, imagine that your toilet bowl is set to empty when it reaches four liters of water inside. So you start pouring water inside and when it reaches the four-liter mark it automatically empties, and the data that contained the water you've poured so far is read or written. But you can also manually empty the toilet by pressing the button that it has. This

causes the toilet to empty and the water (data) inside the toilet is read or written. Just in case you haven't noticed, manually flushing the toilet is a metaphor for hFlush . Maybe this isn't a good analogy in the world of standard programming analogies, but I wanted a real object that could be emptied.

We've already created a program that adds a task to our to-do list todo.txt , so now we're going to create one that removes a task. I'm going to show the code below and then we'll go through the progema together so you see it's really easy. We will use a few new functions found in System.Directory and a new function from System.IO .

Anyway, here is the program that removes a task from todo.txt :

```
import System.IO
import System.Directory
import Data.List

main = do
    handle <- openFile "todo.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\ n line -> show n ++ "-" ++ line) [0 ..] todoTasks
    putStrLn "These are your TO-DO items:"
    putStr $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

First we open the file todo.txt in reading mode and link the manipulator to handle .

Next, we use a function that we don't know yet and that comes from System.IO , openTempFile . Its name is quite self-descriptive. Take the path of a temporary directory and a naming template for a file and open a temporary file. We have used "." for the temporary directory because "." represents the current directory in any OS. We use "temp" as a template for the file name, so that the temporary file will have the name *temp* plus some random characters. Returns an I / O action that creates a temporary file and the result of that action is a pair that contains: the name of the temporary file and the manipulator associated with that file. We could have opened some normal file like todo2.txt

or something like that but it is a best practice to use `openTempFile` and thus make sure we don't overwrite anything.

The reason we haven't used `getCurrentDirectory` to get the current directory and then pass it to `openTempFile` is because `"."` represents the current directory in both *unix* and *Windows systems* .

Then we `link` the contents of `everything.txt` to `contents` . Then we divide that chain into a list of chains, one chain per line. So ` `todoTasks` is now something like `[" Iron the dishes "," Dust the dog ",` `" Take salad out of the oven "]` . We join the numbers from 0 onwards and that list with a function that takes a number, let's say 3, and a string, like `"hey"` , so `numberedTasks` would be `["0 - Iron the dishes", "1 -` Dust the dog "... . We concatenate that list of strings into a single string delimited by line breaks with `unlines` and display it by the terminal. Note that instead of doing this we could have done something like `mapM putStrLn numberedTasks` .

We ask the user which task he wants to delete and we expect him to enter a number. Let's say we want to remove the number 1, which is `Dust the dog` , so we enter `1` . `numberString` is now `"1"` and since we want a number and not a string, we use `read` on it to get a `1` and bind it to `number` .

Try to remember the `delete` and `!!` functions of the `Data.List` module . `!!` returns an element of a list given an index and `delete` removes the first occurrence of an element in a list, and returns a new list without that element. `(todoTasks !! number)` , with `number` to `1` , returns `"Dust the dog"` . We `bind todoTasks` without the first occurrence of `"Dust the dog"` to `newTodoItems` and then join everything into a single string using `unlines` before writing it to the temporary file we've opened. The original file remains unchanged and the temporary file now contains all the tasks that the original contains, except the one that we want to delete.

After closing both files, both the original and the temporary, we remove the original with `removeFile` , which, as you can see, takes the path of a file and removes it. After deleting the original `todo.txt` , we use `renameFile` to rename the temporary file to `todo.txt` . Be `careful` , both `removeFile` and `renameFile` (both contained in `System.Directory`) take file paths and not manipulators.

And that's it! We could have done it in fewer lines, but we are careful not to overwrite any existing files and politely ask the operating system to tell us where we can locate our temporary file. Let's try it!

$ runhaskell deletetodo.hs

```
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
one

$ cat todo.txt
Iron the dishes
Take salad out of the oven

$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0

$ cat todo.txt
Take salad out of the oven
```

Command line parameters

It is practically an obligation to work with command line parameters when we are creating a program that runs in the terminal. Luckily, Haskell's standard library has a good way to get parameters from the command line.

In the previous section, we created a program to add tasks to our to-do list and another program to remove tasks from that list. There are two problems with the approach we take. The first is that we set the name of the file that contained the list in our source code. We simply decided that it would be `todo.txt` and the user could never work with multiple lists.

One way to solve this problem would be to always ask the user which list to work with. We used this approach when we wanted to know what task the user wanted to delete. It works, but there are better options as it requires the user to run the program, wait for the program to ask you something, and then tell you what you need. This is called an interactive program and the problem with interactive programs is: What if we want to automate the execution of the program? As in a batch command file that executes a program or several of them.

For this reason it is sometimes better for the user to tell the program what to do when it runs, rather than for the program to have to ask the user once it

has run. And what better way for the user to tell the program what they want it to do when it is run than with the command line parameters.

The System.Environment module has two very interesting I / O actions. One is getArgs , whose type declaration is getArgs :: IO [String] and it is an I / O action that will obtain the parameters with which the program was executed and the result it contains are those parameters in list form. getProgName has the IO String type and is an I / O action that contains the name of the program.

Here is a small program that demonstrates the behavior of these functions:

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

We link getArgs and getProgName to args and ProgName . We show The arguments are: and then for each parameter in args we do putStrLn . At the end we also show the name of the program. We are going to compile this as arg-test .

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

Well. Armed with this knowledge we can create interesting command line applications. In fact we are going to continue and create one. In the previous section we created separate programs to add and remove tasks. Now we are going to create a program with both functionalities, what you do will depend on the command line parameters. We will also allow that you can work with different files and not only todo.txt .

We will call the program everything and it will be able to do three things:

- See the tasks

- Add a task
- Delete a task

We are not going to worry about possible input errors right now.

Our program will be created so that if we want to add the `Find the magic sword of power` task in the file `todo.txt`, we will have to write `everything add todo.txt "Find the magic sword of power"` in the terminal. To see the tasks we simply run `all view todo.txt` and to delete the task with index 2 we do `all remove todo.txt 2`.

We will start by creating an association list. It will be a simple association list that has the command line parameters and functions as keys as their corresponding values. All these functions will be of type `[String] -> IO ()`. They will take the list of parameters from the command line and return an I / O action that is responsible for displaying tasks, adding a task, or deleting a task.

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch = [("add", add)
            , ("view", view)
            , ("remove", remove)
            ]
```

We have to define `main`, `add`, `view` and `remove`, so let's start with `main`.

```
main = do
    (command: args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

First, we bind the parameters to `(command: args)`. If you remember pattern matching, this means that the first parameter will be bound with `command` and the rest of them with `args`. If we run our program as `all add todo.txt "Spank the monkey"`, `command` will be `"add"` and `args` will be `["todo.txt", "Spank the monkey"]`.

In the next line we look for the command in the association list. Since `"add"` is associated with `add`, we will get `Just add` as a result. We use pattern matching again to extract this function from the `Maybe` type. What if the command was

not in the association list? Well, then it would return `Nothing` , but we have already said that we are not going to worry too much about the errors in the input, so the pattern adjustment would fail and along with it our program.

To finish, we call the `action` function with the rest of the parameter list. This will return an I / O action that will either add a task, or display a list of tasks, or delete a task. And since this action is part of the `do` block of `main` , it will be executed. If we follow the example we have used so far, our `action` function will be `add` , which will be called with `args` (that is, with `["todo.txt", "Spank the monkey"]`) and will return an action that will add the task `Spank the monkey` to `everything.txt` .

Great! All we are left with now is implementing the `add` , `view` and `remove` functions . Let's start with `add` :

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\ n")
```

If we run our program as `all add todo.txt "Spank the monkey"` , `"add"` will be bound to `command` in the first pattern setting of the `main` block , while `["todo.txt", "Spank the monkey"]` will be passed to the function we get from the association list. So since we are not worrying about possible wrong entries, we can use pattern matching directly on a list with those two elements and return an I / O action that adds the task to the end of the file, along with a line break.

Next we will implement the functionality of displaying the task list. If we want to see the elements of a file, we execute `all view todo.txt` . So in the first pattern adjustment, `command` will be `"view"` and `args` will be `["todo.txt"]` .

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\ n line -> show n ++ "-" ++ line) [0 ..] todoTasks
    putStr $ unlines numberedTasks
```

We have already done something very similar in the program that eliminated tasks when displaying tasks so that the user could choose one, only here we only show the tasks.

And to finish we implement `remove` . It will be very similar to the program that eliminated tasks, so if there is something you do not understand, check

the explanation we gave at the time. The main difference is that we do not set the file name to `todo.txt` but we obtain it as a parameter. Neither do we ask the user the index of the task to be eliminated since we also obtain it as one more parameter.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

We open the file based on `fileName` and we open a temporary file, we delete the line with line index that the user wants to delete, we write it in a temporary file, we delete the original file and rename the temporary file to `fileName` .

Here is the entire program in all its glory!

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch = [("add", add)
            , ("view", view)
            , ("remove", remove)
            ]

main = do
    (command: args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\ n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
```

```
    let todoTasks = lines contents
        numberedTasks = zipWith (\ n line -> show n ++ "-" ++ line) [0 ..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

In short: we create an association list that associates the commands with functions that take command line arguments and return I / O actions. We see what command the user wants to execute and we obtain the appropriate function from the association list. We call that function with the rest of the command line parameters and we get an I / O action that will take the appropriate action when it is executed.

In other languages, we should have implemented this using a large switch or anything else, but thanks to higher order functions we are allowed to create an association list that will return the appropriate I / O action for each command we go through. Command line.

Let's try our app!

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from drycleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
```

```
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

Another interesting thing about this is that it is quite easy to add additional functionality. We simply have to add one more element in the association list and then implement the corresponding function. As an exercise you can implement the `bump` command which will take a file and a and an index of a task and will make that task appear at the top of the list of pending tasks.

You can make this program fail more gracefully in case it receives wrong parameters (such as `all UP YOURS HAHAHAH`) by creating an I / O action that simply reports that an error has occurred (say `errorExit :: IO ()`) and then buy if there is any wrong parameter to make the report. Another way would be using exceptions, which we will see shortly.

Randomness

Many times while programming, we need to get some random data. Maybe we are making a game in which you have to roll a die or maybe we need to generate some data to check our program. There are many uses for random data. Well, actually, pseudo-random, since we all know that the only true source of randomness in a monkey on a unicycle with a piece of cheese in one hand and his butt in the other. In this section, we are going to see how Haskell generates apparently random data.

In most other languages, we have functions that return random numbers to us. Every time you call the function, you get (hopefully) a different random number. Well remember, Haskell is a pure functional language. Therefore it has referential transparency. Which means that a function, given the same parameters, must produce the same result. This is great as it allows us to reason about programs in a different way and allows us to delay evaluating operations until we really need them. If we call a function, we can be sure that it won't do anything else before giving us a result. All that matters is your result. However, this makes getting random numbers a bit tricky. If we have a function like:

```
randomNumber :: (Num a) => a
randomNumber = 4
```

It will not be very useful as a function of random numbers since it will always return the same `4`, although I can assure that that 4 is totally random since I just rolled a dice to get it.

What are other languages doing to generate apparently random numbers? Well, first they get some data from your computer, such as the current time, how much and where you have moved the mouse, the noise you make in front of the computer, etc. And based on that, it returns a number that seems random. The combination of those factors (randomness) is probably different at each instant of time, so you get different random numbers.

So in Haskell, we can create a random number if we create a function that takes that randomness as a parameter and returns a number (or any other type of data) based on it.

We will use the `System.Random` module. It contains all the features that will quench our thirst for randomness. We are going to play with one of the functions that it exports, called `random`. Its type declaration is `random :: (RandomGen g, Random a) => g -> (a, g)` Wow! There are new kinds of types in this declaration. The `RandomGen` type `class` is for types that can act as sources of randomness. The `Random` type class is for types that can have random data. A Boolean data can have random values, `True` or `False`. A number can also take a set of different allotary values. Can the function type take random values? I don `t believe. If we translate the `random` type declaration into Spanish we have something like: it takes a random generator (that is, our source of randomness) and returns a random value and a new random generator. Why does it return a new generator next to the random value? We will see it soon.

To use the `random` function, we first have to get one of those random generators. The `System.Random` module exports an `interant` type called `StdGen` that has an instance for the `RandomGen` type `class`. We can create a `StdGen` manually or we can tell the system to give us one based on a bunch of random things.

To manually create a random generator, we use the `mkStdGen` function. It has the type `Int -> StdGen`. It takes an integer and based on that, it returns a random generator. Okay, so let's try using the `mkStdGen` tandem `random` to get a random number.

```
ghci> random (mkStdGen 100)
<interactive>: 1: 0:
    Ambiguous type variable `a 'in the constraint:
```

`Random a 'arising from a use of` random' at <interactive>: 1: 0-20
Probable fix: add a type signature that fixes these type variable (s)

What happens? Ah okay, the `random` function can return any type that is a member of the `Random` type class , so we have to tell Haskell exactly what type we want. Also remember that it returns a random value and a generator.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Finally, a number that seems random! The first component of the pair is our random number while the second component is a textual representation of the new generator. What happens if we call `random again` with the same generator?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Of course. The same result for the same parameters. Let's test it by giving it a different generator parameter.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Great, a different number. We can use type annotation with many other types.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False, 1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

We are going to create a function that simulates launching a modena three times. If `random` didn't return a new generator along with the random value, we would have to make this function take three generators as parameters and then return one result for each of them. But this seems to be not very correct since if a generator can create a random value of type `Int` (which can have a wide variety of possible values) it should be able to simulate three tosses of a coin (which can only have eight possible values). So this is why `random` returns a new generator next to the generated value.

We will represent the result of a coin toss with a simple `Bool` . `True` for face, `False` for cross.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen ') = random newGen
        (thirdCoin, newGen '') = random newGen '
    in (firstCoin, secondCoin, thirdCoin)
```

We call `random` with the generator that we get as a parameter and we get the result of tossing a coin with a new generator. Then we call the same function again, only this time with our new generator, so that we get the second result. If we had called it with the same generator three times, all the results would have been the same and therefore we could only have obtained as results `(False, False, False)` or `(True, True, True)` .

```
ghci> threeCoins (mkStdGen 21)
(True, true, true)
ghci> threeCoins (mkStdGen 22)
(True, False, True)
ghci> threeCoins (mkStdGen 943)
(True, False, True)
ghci> threeCoins (mkStdGen 944)
(True, true, true)
```

Note that we did not have to do `random gen :: (Bool, StdGen)` . It is because we have already specified in the type declaration of the function that we want boolean values. For this reason Haskell can infer that we want boolean values.

And what if we wanted to flip the coin four times? And five? Well, for that we have the function called `randoms` that takes a generator and returns an infinite sequence of random values.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951, -1015194702, -1622477312, -502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True, True, True, True, False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Why does n't `randoms` return a new generator along with the list? We can implement the `randoms` function very simply as:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms 'gen = let (value, newGen) = random gen in value: randoms' newGen
```

A recursive function. We get a random value and a new generator from the current generator and create a list that has the random value as head and a list of random values based on the new generator as tail. Since we want to be able to generate an infinite number of random values, we cannot return a new generator.

We could create a function that would generate finite random number sequences and also return a new generator.

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in (value: restOfList, finalGen)
```

Again, a recursive function. We say that if we want zero random values, we return an empty list and the generator that was given to us. For any other number of random values, we first get a random number and a new generator. This will be the head. Then we say that the queue will be `n-1` generator random values with the new generator. We ended up returning the head together with the rest of the list and the generator that we got when we generated the `n-1` random values.

What if we want to obtain a random value within a certain range? All the integers we have generated so far are outrageously large or small. What if we want to roll a die? Well, for that we use `randomR` . Its type declaration is `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)` , which means it has behavior similar to `random` , only it first takes a pair of values that will set the upper and lower limit so that the generated random value is within that range.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

There is also `randomRs` , which produces a sequence of random values within our range.

```
ghci> take 10 $ randomRs ('a', 'z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Great, it looks like a top secret password.

You may be wondering what this section has to do with I / O. So far we have not seen anything related to I / O. Well, so far we have always created our generator manually based on some arbitrary integer. The problem is, in real programs, they will always return the same random numbers, which is not a very good idea. For this reason System.Random offers us the getStdGen I / O action that has the IO type StdGen . When program execution starts, it asks the system for a good random value generator and stores it in something called a global generator. getStdGen brings that generator so we can link it to something.

Here is a simple program that generates a random string.

```
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a', 'z') gen)
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
Bakzhnnuzrkgvesqplrx
```

Be aware that by calling getStdGen twice we are asking the system twice for the same global generator. If we do something like:

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a', 'z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a', 'z') gen2)
```

We will get the same string displayed twice. One way to get two different strings 20 characters long is to create an infinite list and take the first 20 characters and display them on one line, then take the next 20 and display them on a second line. To do this we use the splitAt of Data.List , which divides a given list and returns a tuple which has the first part as the first component and the second part as the second component index.

```
import System.Random
import Data.List

main = do
    gen <- getStdGen
    let randomChars = randomRs ('a', 'z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

Another way to do this is to use the newStdGen action that splits the random value generator into two new generators. Update the global generator with one of them and the bull returns it as a result of the action.

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a', 'z') gen)
    gen '<- newStdGen
    putStr $ take 20 (randomRs ('a', 'z') gen ')
```

Not only do we get a new generator when we link newStdGen , but the global generator is also updated, so if we use getStdGen later we will get another generator that will be different from gen .

We are going to create a program that makes our user guess the number we are thinking of.

```
import System.Random
import Control.Monad (when)

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of?"
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was" ++ show randNumber
        askForNumber newGen
```

We have created the `askForNumber` function , which takes a random value generator and returns an I / O action that will ask the user for a number and tell him whether or not he was right. Within this function, we first generate a random number and new generator based on the generator we got as a parameter, we call them `randNumber` and `newGen` . Let's say the generated number is `7` . Then we ask the user what number we are thinking about. We run `getLine` and `bind` the result to `numberString` . When the user enters `7` , `numberString` becomes `"7"` . Then we use a `when` clause to check if the string entered by the user is empty. If it is, an empty I / O action (`return ()`) will be executed, thus ending our program. If it is not, the action contained in the `do` block will be executed. We use `read` over `numberString` to convert it to a number, which will now be `7` .

Note

If the user enters something that `read` can't read (like `"haha"`), our program will abruptly end with a pretty horrendous error message. If you don't feel like the program ending this way, use the `reads` function , which returns an empty list when it can't read a string. When if it returns a unit list that contains a pair with our desired value as the first component and a string with what has not been consumed as the second component.

We check if the number they have entered is equal to the number we have randomly generated and give the user an appropriate message. Then we call `askForNumber` recursively, only this time with the new generator that we have obtained, so that we get an I / O action like the one we just executed, only it depends on a different generator.

`main` is basically getting the random value generator and calling `askForNumber` with the initial generator.

Here is our program in action!

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? two
Sorry, it was 4
```

```
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

Another way to do the same program would be:

```
import System.Random
import Control.Monad (when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of?"
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was" ++ show randNumber
        newStdGen
        main
```

It is very similar to the previous version, only instead of doing a function that takes a generator and then calls itself recursively, we do all the work in `main`. After telling the user if the number they thought is correct or not, we update the global generator and call `main again`. Both implementations are valid but I like the first one more since the `main` performs fewer actions and also provides us with a function that we can reuse.

Byte strings

Lists are great data structures as well as useful. Until now we have used them anywhere. There are a multitude of functions that work with them and Haskell's lazy evaluation allows us to swap them for loops when filtering and plotting, as evaluation only happens when really needed, so infinite lists (even infinite lists of infinite lists!) are not a problem for us. For this reason, lists can also be used to represent data streams, either to read from standard input or from a file. We can open a file and read it as if it were a string, even if it is only accessed as far as our needs reach.

However, processing files as strings has a drawback: it is usually slow. As you know, `String` stands for type of `[Char]`. `Char` does not have a fixed size, as

it can take multiple bytes to represent a character. Also, the lists are lazy. If you have a list like `[1,2,3,4]`, it will be evaluated only when completely necessary. So the entire list is a kind of promise that at some point it will be a list. Remember that `[1,2,3,4]` is simply a syntactic decoration for `1: 2: 3: 4: []`. When the first item in the list is forced to evaluate (say, by displaying it on the screen), the rest of the list `2: 3: 4: []` remains a promise of a list, and so on. So you can think of the lists as if they were promises that the next item will be delivered once needed. It doesn't take much thought to conclude that processing a simple list of numbers as a series of promises should not be the most efficient thing in the world.

This overload does not usually worry us most of the time, but it should when reading and manipulating large files. For this reason Haskell has **byte strings**. Byte strings are a kind of lists, only each element is the size of a byte (or 8 bits). The way they are evaluated is also different.

There are two types of byte strings: strict and lazy. The strict ones reside in `Data.ByteString` and it doesn't have any lazy evaluation. There is no promise involved, a strict byte string represents a series of bytes in a vector. We cannot create things like infinite byte chains. If we evaluate the first byte of a strict byte string we evaluate the entire string. The advantage is that there is less overhead since it does not imply any *thunk* (technical term of *promise*). The downside is that they will consume memory much faster since they are read into memory in one fell swoop.

The other type of byte strings resides in `Data.ByteString.Lazy`. They are lazy, but not in the same way as the lists. As we have already said, there are as many *thunks* as there are elements in a normal chain. This is why they are slow in some situations. Lazy byte strings take another approach, they are stored in 64KB blocks in size. In this way, if we evaluate a byte in a lazy byte string (showing it on screen or something similar), the first 64KB will be evaluated. After these, there is only one promise that the following will be evaluated. Lazy byte strings are kind of like a 64KB byte string list. When we process files using lazy byte strings, the file contents will be read block by block. It is great since it will not take the memory to its limits and probably 64KB will fit perfectly in the L2 cache memory of your processor.

If you look at the documentation of `Data.ByteString.Lazy`, you will see that exports a lot of functions that have the same name as those of `Data.List`, just type in their statements have `ByteString` instead of `[a]` and `WORD8` of `a` from

inside. Functions with similar names behave practically the same except that some work with lists and others with byte strings. Since they import equal function names, we are going to import them in a qualified way in our code and then we will load it into GHCi to play with the byte strings.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` has lazy byte strings while `S` contains strict ones. We will almost always use the lazy version.

The `pack` function has a type `[Word8] -> ByteString` . Which means it takes a list of bytes of type `Word8` and returns a `ByteString` . You can see it as taking a list, which is lazy, and makes it less lazy, so it's still lazy only at 64KB intervals.

What about the `Word8` type ? Well, it's like `Int` , only it has a much smaller range, from 0 to 255. It represents a number of 8b. And like `Int` , it is a member of the `Num` class . For example, we know that the value 5 is polymorphic since it can behave like any numeral type. Well, you can also take the `Word8` type .

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

As you can see, you usually don't have to worry much about the `Word8` type , since the type system can make the numbers take that type. If you try to use a very large number, like 336, like a `Word8` , it will just be binary truncated to the value 80.

We have packed only a few values within a byte string, so that they fit within the same block (`Chunk`). The `Empty` is like `[]` for lists.

`unpack` is the reverse version of de `pack` . It takes a string of bytes and converts it to a list of bytes.

`fromChunks` takes a list of strict byte strings and converts it to a lazy byte string. `toChunks` takes a lazy byte string and turns it into a strict one.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "() *" (Chunk "+, -" (Chunk "./0" Empty))
```

This is useful when you have a bunch of strict byte strings and want to

process them efficiently without having to merge them into a larger memory first.

The version of `:` for byte strings is known as `cons`. Take a byte and a byte string and put that byte at the beginning. Although it is lazy, it will generate a new block for that item even though that block is not yet full. For this reason it is better to use the strict version of `cons`, `cons'`, if you are going to insert a bunch of bytes at the beginning of a byte string.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk
"9" (Chunk ":" (Chunk ";" Chunk "<" Empty)))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:; <" Empty
```

As you can see `empty` create an empty byte string Can you see the differences between `cons` and `cons'` ? With the help of `foldr` we have started with an empty byte string and then we have `scrolled` through the list of numbers from the right, adding each number to the beginning of the byte string. When we use `cons`, we end up with one block for each byte, which is not very useful for our purposes.

Either way, byte string modules have a ton of functions analogous to `Data.List`, including, but not limited to, `head`, `tail`, `init`, `null`, `length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

They also contain functions with the same name and behavior as some functions found in `System.IO`, only `String` is replaced by `ByteString`. For example, the function `readFile` of `System.IO` has the type `readFile :: FilePath -> IO String`, while `readFile` modules byte strings have the type `readFile :: FilePath -> IO ByteString`. Be careful, if you are using the strict version of byte strings and trying to read a file, it will be read into memory in one fell swoop. Lazy byte strings will read block by block.

We are going to create a simple program that takes two file paths as command line parameters and copies the contents of the first to the second. Note that `System.Directory` already contains a function called `copyFile`, but we are going to implement our program like this anyway.

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1: fileName2: _) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

We create our own function that takes two `FilePaths` (remember, `FilePath` is just a synonym for `String`) and returns an I / O action that will copy the contents of a file using byte strings. In the `main` function , we simply get the parameters and call our function with them to get an I / O action that will be executed.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Note that a program that does not use byte strings may have the same similarity, the only difference would be that instead of writing `B.readFile` and `B.writeFile` we `would` use `readFile` and `writeFile` . Many times we can convert a program that uses strings to a program that uses byte strings simply by using the correct modules and qualifying some functions. Sometimes, they may need to convert string-working functions to work with byte strings, but it's not too difficult.

Whenever you need higher performance in programs that read lots of data in the form of strings, try using byte strings, you have a good chance of achieving higher performance with very little effort. Normally I usually create programs that work with normal strings and then convert them to byte strings because the performance is not on target.

Exceptions

All languages have procedures, functions, or bits of code that fail in some way. It is a law of life. Different languages have different ways of handling these bugs. In *C* , we usually use an abnormal return value (such as -1 or a null pointer) to indicate that the returned value should not be handled normally. *Java* and *C #* , on the other hand, tend to use exceptions to handle these flaws. When an exception is thrown, the code execution jumps to

somewhere we've defined to perform the appropriate tasks and may even re-launch the exception so that it is handled elsewhere.

Haskell has a good type system. Algebraic data types allow us to have types like `Maybe` and `Either` that we can use to represent results that are valid and that are not. In *C* , returning, say -1, when an error occurs is a matter of convention. It only has special meaning for humans. If we are not careful, we can treat these abnormal data as valid so that our code ends up being a real disaster. Haskell's type system gives us the security we need in this regard. A function `a -> Maybe b` clearly indicates that it can produce a `b` wrapped by a `Just` or it can return `Nothing` . The type is completely different from `a -> b` and if we try to use these two functions interchangeably, the type system will complain.

Although still having expressive types that support erroneous operations, Haskell still has support for exceptions, they already make more sense in the context of I / O. A lot of things can go wrong when we are dealing with the outside world as it is not very reliable. For example, when we open a file, quite a few things can go wrong. The file may be protected, it may not exist or there may not even be a physical medium for it. So it's okay to be able to jump somewhere in our code to deal with an error when that error happens.

Okay, so I / O code (i.e. impure code) can throw exceptions. It makes sense. But what about pure code? Well, you can also throw exceptions. Think of the `div` and `head` functions . They have the types `(Integral a) => a -> a ->` and `[a] -> a` respectively. There is no `Maybe` or `Either` in the type they return but they may still fail. `div` may fail if you try to divide something by zero and `head` when you give it an empty list.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Pure code can throw exceptions, but they can only be caught in the I / O parts of our code (when we are inside a `do` block that is reached by `main`). This happens because we do not know when (or if) something will be evaluated in the pure code since it is lazily evaluated and does not have a specific execution order defined, while the I / O parts do.

Before we talked about how we should stay as short as possible in the I / O

parts of our program. The logic of our program must remain mostly in our pure functions, since its results only depend on the parameters with which we call them. When you are dealing with pure functions, we just have to worry that it returns a function, since it can't do anything else. This makes our life easier. Although performing some tasks on the I / O part is essential (like opening a file and the like), they should be kept to a minimum. Pure functions are lazy by default, which means we don't know when they will be evaluated and we really don't need to worry either. However, when pure functions start throwing exceptions, it does matter when they are evaluated. For this reason we can only catch exceptions thrown from pure code in the I / O parts of our program. And since we want to keep the I / O parts to a minimum this doesn't benefit us much. However, if we don't capture them in an I / O part of our code, the program will abort Solution? Don't mix exceptions with pure code. Take advantage of Haskell's powerful type system and use types like `Either` and `Maybe` to represent results that may be wrong.

For this reason, for now we will only see how to use I / O exceptions. I / O exceptions occur when something goes wrong when communicating with the outside world. For example, we can try to open a file and then it can happen that that file has been deleted or something similar. Look at the following program, which opens a file that has been obtained as a parameter and tells us how many lines it contains.

```
import System.Environment
import System.IO

main = do (fileName: _) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has" ++ show (length (lines contents)) ++ "lines!"
```

A very simple program. We perform the `getArgs` I / O `action` and `bind` the first string in the string that returns us to `fileName` . Then we call the file `contents` as `contents` . To finish, we apply `lines` to those contents to get a list of lines and then we get the length of that list and display it using `show` . It works as expected, but what happens when we give it the name of a file that does not exist?

```
$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)
```

AHA! We get a *GHC* error that tells us that this file does not exist. Our program fails. What if we wanted to display a more pleasant message in case

the file does not exist? One way to do this would be to check if the file exists before trying to open it using the `doesFileExist` function of `System.Directory` .

```
import System.Environment
import System.IO
import System.Directory

main = do (fileName: _) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
               then do contents <- readFile fileName
                       putStrLn $ "The file has" ++ show (length (lines contents)) ++ "lines!"
               else do putStrLn "The file doesn't exist!"
```

We made `fileExists <- doesFileExist fileName` because `doesFileExist` has a type declaration of `doesFileExist :: FilePath -> IO Bool` , which means that it returns an I / O action that results in a boolean value that tells us whether the file exists or not. We cannot use `doesFileExist` directly in an `if` expression .

Another solution would be using exceptions. It is perfectly acceptable to use them in this context. A file that does not exist is an exception that is thrown from I / O, so catching it on I / O is totally acceptable.

To deal with this using exceptions, let's take advantage of the `catch` function in `System.IO.Error` . Its type declaration is `catch :: IO a -> (IOError -> IO a) -> IO a` . Take two parameters. The first is an I / O action. For example, it could be an action that tries to open a file. The second is what we call a manipulator. If the first I / O action we pass to `catch` throws an exception, the exception passes to the handler that decides what to do. So the end result will be an action that will either act as its first parameter or do what the handler says in case the first I / O action throws an exception.

If you are familiar with *try-catch* blocks in languages like *Java* or *Python* , the `catch` function is similar to them. The first parameter is what to try to do, something like what is inside a *try* block . The second parameter is the handler that takes an exception, in the same way that most *catch* blocks take exceptions that you can examine to see what has happened. The manipulator is invoked if an exception is thrown.

The handler takes a value of type `IOError` , which is a value that represents an I / O exception has occurred. They also contain information about the exception that has been thrown. The implementation of this type depends on

the implementation of the language itself, so we cannot inspect values of the IOError type using the adjustment of patterns on them, in the same way that we cannot use the adjustment of patterns with values of the IO type . However, we can use a bunch of useful predicates to examine the IOError type values as we will see in a few seconds.

So let's put our new catch friend to use .

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName: _) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has" ++ show (length (lines contents)) ++ "lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

First of all, you can see how we have used single quotes to use this function in an infix way, since it takes two parameters. Using it infixed makes it more readable. So toTry `catch` handler is the same as catch toTry handler , plus it matches its type. toTry is an I / O action that we will try to execute and handler is the function that takes an IOError and returns an action that will be executed in the event of an exception.

We're going to try it:

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!

$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

We have not verified what type of IOError we get inside the handler . We just say "Whoops, had some trouble!" for any kind of error. Catching all types of exceptions by the same handler is not a good practice in Haskell or in any other language. What if some other exception that we don't want to catch is thrown, such as if we interrupt the program or something similar? For this reason we are going to do the same thing that is usually done in other languages: we will check what kind of exception we are catching. If the exception is the type we want to catch, we will do our job. If not, we will

relaunch that same exception. We are going to modify our program so that it only catches the exceptions due to the fact that a file does not exist.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName: _) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has" ++ show (length (lines contents)) ++ "lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | otherwise = ioError e
```

Everything remains the same except the manipulator, which we have modified so that it only catches a group of I / O exceptions. We have used two new functions from `System.IO.Error` , `isDoesNotExistError` and `ioError` . `isDoesNotExistError` is a predicate on `IOError` , or what is the same, it is a function that takes a value of type `IOError` and returns `True` or `False` , so its type declaration is `isDoesNotExistError :: IOError -> Bool` . We have used this function with the exception that is passed to the manipulator to see if the error was due to the fact that there was no file. We also use the *guard* syntax , although we could have used an `if else` . In case the exception was not thrown due to a file not being found, we relaunched the exception that was passed to the handler using the `ioError` function . Its type declaration is `ioError :: IOException -> IO a` , so it takes an `IOError` and produces an I / O action that throws that exception. The I / O action has the `IO` type `a` since it will never really return a value.

In short, if the exception thrown within the I / O `toTry action` that we have included within the `do` block is not because there is no file, `toTry `catch` handler will` catch that exception and throw it again.

There are several predicates that work with `IOError` that we can use together the guards, since if a guard is not evaluated to `True` , the next guard will continue to be evaluated. The predicates that work with `IOError` are:

- isAlreadyExistsError
- isDoesNotExistError
- isAlreadyInUseError

- isFullError
- isEOFError
- isIllegalOperation
- isPermissionError
- isUserError

Most of these are self explanatory. `isUserError` evaluates to `True` when we use the `userError` function to create the exception, which is used to create exceptions in our code and accompany them with a string. For example, you can use something like `ioError $ userError "remote computer unplugged!"` , although it is preferable that you use the `Either` and `Maybe` types to represent possible failures instead of throwing exceptions yourself with `userError` .

We could have a manipulator that looked something like this:

```
handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | isFullError e = freeSomeSpace
    | isIllegalOperation e = notifyCops
    | otherwise = ioError e
```

Where `notifyCops` and `freeSomeSpace` are I / O actions that we have defined. Be sure to re-throw exceptions that don't meet your criteria, otherwise you will stealthily crash your program when it shouldn't.

`System.IO.Error` also exports some functions that allow us to ask these exceptions for some attributes, such as which handler caused the error, or what file path caused it. These functions start with `ioe` and you can see the full list in the documentation. Let's say we want to show the path of a file that caused an error. We cannot display the `fileName` that we got from `getArgs` , since only a value of type `IOError` is passed to the handler and the handler knows nothing else. A function depends exclusively on the parameters with which it was called. For this reason we can use the `ioeGetFileName` function , whose type declaration is `ioeGetFileName :: IOError -> Maybe FilePath` . Take an `IOError` as a parameter and maybe return a `FilePath` (which is a synonym for `String` , so it's pretty much the same). Basically what it does is extract the path of a file from an `IOError` , if you can. We are going to modify the previous program so that it shows the path of the file that caused a possible exception.

```
import System.Environment
```

```
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName: _) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has" ++ show (length (lines contents)) ++ "lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e =
        case ioeGetFileName e of Just path -> putStrLn $ "Whoops! File does not exist at:" ++
path
                                 Nothing -> putStrLn "Whoops! File does not exist at unknown
location!"
    | otherwise = ioError e
```

If the save where isDoesNotExistError is found evaluates to True , we use a case expression to call ioeGetFileName with e and apply a pattern wrapper with the Maybe that returns. We normally use case expressions when we want to apply a pattern fit without having to create a new function.

You don't have to use a handler to catch all the exceptions that occur in the I / O part of your program. You can cover certain parts of your I / O code with catch, or you can cover several of them with catch and use different manipulators. Something like:

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Here, toTry uses handler1 as a manipulator and thenTryThis uses handler2 . launchRockets is not a parameter of any catch , so any exceptions thrown will abort the program, unless launchRockets internally uses a catch that handles its own exceptions. Of course toTry , thenTryThis and launchRockets are I / O actions that have been attached with a do block and hypothetically defined somewhere. It is similar to *try-catch* blocks that appear in other languages, where you can use a single *try-catch* block to wrap the entire program, or you can use a more detailed approach and use different blocks in different parts of the program.

Now you know how to deal with I / O exceptions. We have not seen how to throw exceptions from pure code and work with them, because, as we have already said, Haskell offers better ways to report errors without resorting to

parts of the I / O. Even though I have to work with I / O actions that can fail, I prefer to have types like `IO (Either ab)`, which indicate that they are normal I / O actions only that their result will be of type `Either ab`, so either will return `Left a` or `Right b`.