Haskell for Readers

Joachim Breitner[*], DFINITY Foundation[†]

Contents

[*] http://www.joachim-breitner.de/
[†] https://dfinity.org/

Preface

Welcome to the lecture series "Haskell for Readers". This workshop is uniquely tailored to those who need to *read*, rather than *write* Haskell code: auditors, scientists, managers, testers etc.

Scope

This goal implies a higher focus on syntax (because you can *write* programs ignoring most syntactic gadgets available to you, but if you read code, you have to deal with them), types and type signatures (because they are the key to understanding Haskell code) and abstraction patterns (because it is key to understanding well-written code, and Haskell excels at abstraction).

On the other hand, less words will be spent on how to approach writing the program, e.g. how to set up your tooling, how to please Haskell's layout rules, how to design your data type, which libraries to pick, how to read error messages. That said, we hope that even Haskell programmers will gain useful insight from this tutorial.

Nevertheless it is hard to understand a programming paradigm without writing any code, so there will some amount of hands-on work to be done, especially early on, when we start with a introduction to basic functional programming.

Form

This document is not (necessarily) a self-contained tutorial; it is rather the base for an interactive lecture, given by a real instructor. In such a lecture, some holes will be filled as we go, and the questions from the audience form a crucial part of the learning experience.

This document is also meant to be more on the concise side, assuming the audience is shorter on time than on wits, and in a small, live workshop, the lecturer can add details, come up with more examples and slow down as needed.

This makes these notes less ideal for independent study, but that said, it should be possible to work attentively through them and still learn a lot.

There is a printable versions of the whole document[1] (with solutions at the end), of just the exercises[2] and of just the solutions[3].

Audience

I expect the audience to be familiar with programming and computer science in general, but do not assume prior knowledge of functional program (or, in case you are worried about this, category theory).

The exercises are all very small, in the order of minutes, and are meant to be done along the way, especially as later material may refer to their results. If you are reading this on your own and you really do not feel like doing them, you can click on the blurred solutions to at least read them. The exercises are not sufficient in number and depth to provide the reader with the experience needed to really learn Haskell.

Some sections are marked with a "⋆". These are optional in the sense that the following material does not rely heavily on them. If time is short, e.g. during a workshop, they can be skipped, and the participants can be invited to come back to them on their own.

Acknowledgments and license

The creation of this material was sponsored by the DFINITY Foundation[4], and is shared with the public under the terms of the Creative Commons Attribution 4.0 International License[5]. You can view the source

[1] http://haskell-for-readers.nomeata.de/haskell-for-readers.pdf
[2] http://haskell-for-readers.nomeata.de/haskell-for-readers-exercises.pdf
[3] http://haskell-for-readers.nomeata.de/haskell-for-readers-solutions.pdf
[4] http://dfinity.org/
[5] https://creativecommons.org/licenses/by/4.0/

on GitHub[6] of this document, and submit improvements there.

1 The basics of functional programming

Functional programming is the the art of thinking about *data* and how the new data is calculated from old data, rather than thinking about how to *modify* data.

1.1 Numbers and arithmetic operators

The simplest form of data are numbers, and basic arithmetic is one way of creating new numbers from old numbers.

To play around with this, start the Haskell REPL ("read-eval-print-loop") by running `ghci` (or maybe on tryhaskell.org[7]), and enter some numbers, and some of the usual arithmetic operations:

```
$ ghci
GHCi, version 8.4.4: http://www.haskell.org/ghc/  :? for help
Prelude> 1
1
Prelude> 1 + 1
2
Prelude> 2 + 3 * 4
14
Prelude> (2 + 3) * 4
20
```

At this point we can tell that the usual precedence rules apply (i.e. the PEMDAS rule[8]).

```
Prelude> 0 - 1
-1
```

Numbers can be negative...

```
Prelude> 2^10
1024
Prelude> 2^2^10
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
Prelude> (2^2)^10
1048576
```

...and also very large. By default, Haskell uses arbitrary precision integer arithmetic. Note that for this lecture, we will completely avoid and ignore floating point arithmetic.

[6] https://github.com/nomeata/haskell-for-readers/
[7] https://tryhaskell.org/
[8] https://en.wikipedia.org/wiki/Order_of_operations#Mnemonics

In the last example we can see that Haskell interprets `a^b^c` as `a^(b^c)`, i.e. the power operator is *right associative*. It is worth noting that this information is not hard-coded in the compiler. Instead, when the operator is defined somewhere in a library, its associativity and precedence can be declared. We can ask the compiler about this information:

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a   -- Defined in ‘GHC.Real’
infixr 8 ^
```

Let us ignore the first line (which is the type signature): The `r` in `infixr` tells us that the `(^)` operator is right-associative. And the number is the precedence; a higher number means that this operator binds more tightly.

Exercise 1

What associativity do you expect for `(+)` and `(-)`? Verify your expectation.

Exercise 2

Look up the precedences of the other arithmetic operations, and see how that corresponds to the PEMDAS rule.

1.2 Applying Functions

So far we have a calculator (which is not useless, I sometimes use `ghci` as a calculator). But to get closer to functional programming, let us look at some functions that are already available to use.

To stay within the realm of arithmetic (if only to have something to talk about), let us play with the `div` and `mod` functions. These do what you would expect from them:

```
Prelude> div 123 100
1
Prelude> mod 123 100
23
```

We observe that to apply a function, we just write the function, followed by the arguments; no parentheses or commas needed. This not only makes for more elegant and less noisy code; but there is also a very deep and beautiful reason for this, which we will come to later.

At this point, surely someone wants to know what happens when we divide by 0:

```
Prelude> div 123 0
*** Exception: divide by zero
```

Haskell has exceptions, they can even be caught etc., but we will not talk about that for now.

Of course, if the argument is not just a single number, we somehow have to make clear where the argument begins and ends:

```
Prelude> div (120 + 3) (10 ^ 2)
1
```

(If you leave out the parentheses, you get a horrible error message.) In technical terms, we can say that function application behaves like a left-associative operator of highest precedence. But it is easier to just

remember **function application binds most tightly**. (Exception: Record construction and update binds even more tightly, although some consider that a design flaw.)

Just to have more examples, here are two other functions that we can play around with:

```
Prelude> id 42
42
Prelude> const 23 42
23
```

Exercise 3

Can you predict the result of the following?

```
Prelude> 1 + const 2 3 + 4
```

A note on syntactic sugar: Haskell is a high-calorie language: There is lots of syntactic sugar. Syntactic sugar refers to when there are alternative ways of writing something that *look* different, but *behave* the same. The goal is to allow the programmer to write the code in a way that best suits the reader, which is good, but it also means that a reader needs to know about the sugar.

Infix operator application (syntactic sugar): Functions that take two arguments can be written infix, as if they were an operator, by putting backticks around the name:

```
Prelude> 123 `div` 10
12
Prelude> 123 `mod` 10
3
Prelude> (120 + 3) `div` (10 ^ 2)
1
Prelude> (120 + 3) `div` 10 ^ 2
1
Prelude> ((120 + 3) `div` 10) ^ 2
144
```

We see that written as operators, even functions have an associativity and precedence:

```
Prelude> :info div
class (Real a, Enum a) => Integral a where
  ...
  div :: a -> a -> a
  ...
    -- Defined in ‘GHC.Real’
infixl 7 `div`
```

Prefix operator application (syntactic sugar) ✯: We can also go the other way, and use any operator as if it were a function, by wrapping it in parentheses:

```
Prelude> 1 + 1
2
Prelude> (+) 1 1
2
```

The dollar operator (non-syntactic sugar) ⋆: Consider an expression that takes a number, and applies a number of functions, maybe with arguments, to it, such as:

```
f5 (f4 (f3 (f2 (f1 42))))
```

Passing a piece of data through a number of functions is very common, and some (including me) greatly dislike the accumulation of parentheses there. Therefore, it is idiomatic to use the `($)` operator:

```
f5 $ f4 $ f3 $ f2 $ f1 42
```

This operator takes a function as the first argument, an argument as the second argument, and applies the function to the argument. In that way, it is exactly the same as function application. But it is *right-associative* (instead of left-associative) and has the *lowest precedence* (instead of the highest precedence). An easier way of reading such code is to read `($)` as "the same as parentheses around the rest of the line".

I call this non-syntactic sugar, because the dollar operator is not part of the built-in language, but can be defined by anyone.

Exercise 4

What is the result of

```
Prelude> (-) 5 $ div 16 $ (-) 10 $ 4 `div` 2
```

1.3 Booleans and branching

Some cryptographers might be happy to only write code that always does the same thing (yay, no side effects), but most of us pretty quickly want to write branching code.

As you would expect, Haskell has the usual operators to compare numbers:

```
Prelude> 1 + 1 == 2
True
Prelude> 4 < 5
True
Prelude> 4 >= 5
False
Prelude> 23 /= 42
True
```

We see that there are values `True` and `False`. We can combine them using the usual Boolean operators:

```
Prelude> 1 + 1 == 2 && 5 < 4
False
Prelude> 1 + 1 == 2 || 5 < 4
True
```

And finally, we can use `if … then … else …` to branch based on such a Boolean expression:

```
Prelude> if 5 < 0 then 0 else 1
1
Prelude> if -5 < 0 then 0 else 1
0
```

The use of `if … then … else …` is actually not the most idiomatic way to code decisions in Haskell, and we will come back to that point later, but for now it is good enough.

1.4 Function abstraction

Assume you want to check a bunch of numbers as to whether they are multiples of 10 (so called "round numbers" in German). You can do that using `mod` and `(==)`:

```
Prelude> 5 `mod` 10 == 0
False
Prelude> 10 `mod` 10 == 0
True
Prelude> 11 `mod` 10 == 0
False
Prelude> 20 `mod` 10 == 0
True
Prelude> -20 `mod` 10 == 0
True
Prelude> 123 `mod` 10 == 0
False
```

But this gets repetitive quickly. And whenever we program something in a repetitive way, we try to recognize the *pattern* and abstract over the changing *parameter*, leaving only the common parts

Here, the common pattern is `x `mod` 10 == 0`, with a parameter named `x`. We can give this pattern a name, and use it instead:

```
Prelude> isRound x = x `mod` 10 == 0
Prelude> isRound 5
False
Prelude> isRound 10
True
Prelude> isRound 11
False
Prelude> isRound 20
True
Prelude> isRound (-20)
True
Prelude> isRound 123
False
```

And now we are squarely in the realm of functional programming, as we have just defined out first function, `isRound`!

Note that we defined the `isRound` by way of an equation. And it really is an equation: Wherever we see `isRound something`, we can obtain its meaning by replacing it with `something `mod` 10 == 0`. This *equational reasoning*, where you replace equals by equals, is one key technique to make sense of Haskell programs.

Exercise 5

Discuss: Think of other programming language that have concepts called functions. Can you always replace a function call with the function definition? Does it change the meaning of the program?

Exercise 6

Write a function `absoluteValue` with one parameter. If the parameter is negative, returns its opposite number, otherwise the number itself.

Exercise 7

Write a function `isHalfRound` that checks if a number is divisible by 5, by checking whether the last digit is 0 or 5.

Exercise 8

Write a function `isEven` that checks if a number is divisible by 2, by checking whether the last digit is 0, 2, 4, 6, 8.

Of course, you can abstract over more than one parameter. In the last exercise, you had to write something like `x `mod` 10 == y` a lot. So it makes sense to abstract over that:

```
Prelude> hasLastDigit x y = x `mod` 10 == y
```

This allows us to define `isHalfRound` as follows:

```
Prelude> isHalfRound x = x `hasLastDigit` 0 || x `hasLastDigit` 5
```

which, if you read it out, is almost a transliteration of the specification! Here we see how abstraction, together with good naming and syntax, can produce very clear and readable code.

Infix operator application again (syntactic sugar): By the way, you can use infix operator syntax already when defining a function:

```
x `divides` y = x `div` y == 0
```

1.5 Recursion

We already saw that one function that we defined could call another. But the real power of general computation comes when a function can call itself, i.e. when we employ recursion. Recursion is a very fundamental technique in functional programming, much more so than loops or iterators or such.

Let us come up with a function that determines the number of digits in a given number. We first check if the number is already just one digit:

```
Prelude> countDigits n = if n < 10 then 1 else ▉
```

At this point, we know that the number is larger than 10. So to count the digits, we would like to cut off one digit:

```
Prelude> countDigits n = if n < 10 then 1 else (n `div` 10)▉
```

and count the number of digits of *that* number

```
Prelude> countDigits n = if n < 10 then 1 else countDigits (n `div` 10)▉
```

and, of course, add one to that number:

```
Prelude> countDigits n = if n < 10 then 1 else countDigits (n `div` 10) + 1
Prelude> countDigits 0
1
Prelude> countDigits 5
1
Prelude> countDigits 10
2
Prelude> countDigits 11
2
Prelude> countDigits 99
2
Prelude> countDigits 100
3
Prelude> countDigits 1000
4
Prelude> countDigits (10^12345)
12346
```

The fact that we can replace equals with equals does not change just because we use recursion. For example, we can figure out what `countDigits 789` does by replacing equals with equals:

```
  countDigits 789
= if 789 < 10 then 1 else countDigits (789 `div` 10) + 1
= if False then 1 else countDigits (789 `div` 10) + 1
= countDigits (789 `div` 10) + 1
= countDigits 78 + 1
= (if 78 < 10 then 1 else countDigits (78 `div` 10) + 1) + 1
= (if False then 1 else countDigits (78 `div` 10) + 1) + 1
= (countDigits (78 `div` 10) + 1) + 1
= (countDigits 7 + 1) + 1
= ((if 7 < 10 then 1 else countDigits (7 `div` 10) + 1) + 1) + 1
= ((if True then 1 else countDigits (7 `div` 10) + 1) + 1) + 1
= (1 + 1) + 1
= 2 + 1
= 3
```

Exercise 9

Write the function `sumDigits` that sums up the digits of a natural number.

1.6 Higher-order functions

We created functions when we took expressions that followed a certain pattern, and abstracted over a number that occurred therein. But the thing we can abstract over does not have to be just a simple number. It could also be a function!

Consider the task of calculating the number of digits in the number of digits of a number:

```
Prelude> countDigits (countDigits 5)
1
```

```
Prelude> countDigits (countDigits 10)
1
Prelude> countDigits (countDigits (10^10))
2
Prelude> countDigits (countDigits (10^123))
3
Prelude> countDigits (countDigits (15^15))
2
```

Clearly, we can abstract over the argument here:

```
Prelude> countCountDigits n = countDigits (countDigits n)
Prelude> countCountDigits (10^123)
3
```

But now consider we also want `sumSumDigits`:

```
Prelude> sumSumDigits n = sumDigits (sumDigits n)
Prelude> sumSumDigits (9^9)
9
Prelude> sumSumDigits (7^7)
7
Prelude> sumSumDigits (13^13)
13
Prelude> sumSumDigits (15^15)
18
```

There is clearly a pattern that is shared by both `countCountDigits` and `sumSumDigits`: They both apply a function twice. And indeed, we can abstract over that pattern:

```
Prelude> twice f x = f (f x)
Prelude> twice countDigits (15^15)
2
Prelude> twice sumDigits (15^15)
18
```

This is our first *higher order function*, and it is called so because it is a function that take another function as an argument. More precisely, it is called a second-order function, because it takes a normal, i.e. first-order function, as an argument. Abstracting over a second order function yields a third order function, and so on. Up to sixth-order functions[9] are seen in the wild.

If you look at the last two lines, we again see a common pattern. And abstracting over that, we recover very nice and declarative definitions for `countCountDigits` and `sumSumDigits`:

```
Prelude> countCountDigits x = twice countDigits x
Prelude> sumSumDigits x = twice sumDigits x
```

The ability to abstract very easily over functions is an important ingredient in making Haskell so excellent at abstraction: It allows to abstract over *behavior*, instead merely over *value*. To demonstrate that, let us recall the definitions of `countDigits` and `someDigits`:

```
Prelude> countDigits n = if n < 10 then 1 else countDigits (n `div` 10) + 1
```

[9] https://doi.org/10.1017/S0956796898003001

```
Prelude> sumDigits n = if n < 10 then n else sumDigits (n `div` 10) + (n `mod` 10)
```

These two functions share something: They share a behavior! Both iterate over the digits of the function, do something at each digit, and sum something up. And this common functionality is not trivial! So it is very unsatisfying to copy'n'paste it, like we did. So how can we abstract over the parts that differ? It is not obvious on first glance, so go through it step by step, lets give the parts that differ names. For `countDigits`, we *ignore* the digit, and just sum up ones:

```
Prelude> always1 n = 1
Prelude> countDigits n = if n < 10 then always1 n else countDigits (n `div` 10) + always1 (n `mod` 10)
```

And for `sumDigits`, we use the digit as is. And we have already seen a function that just returns its argument, the identity function:

```
Prelude> sumDigits n = if n < 10 then id n else sumDigits (n `div` 10) + id (n `mod` 10)
```

Now the common pattern is clear, and we can abstract over the different parts:

```
Prelude> sumDigitsWith f n = if n < 10 then f n else sumDigitsWith f (n `div` 10) + f (n `mod` 10)
Prelude> countDigits n = sumDigitsWith always1 n
Prelude> sumDigits n = sumDigitsWith id n
```

To recapitulate: We took two functions that were doing somehow related things, and we rewrote them to clearly separate the common parts from the differing parts, and then we could extract the shared essence into its own, higher-order function.

This single mechanism – abstracting over functions – can replace thick volumes full of design patterns[10] in non-functional programming paradigms.

Note that if one would have to abstract `countDigits` and `sumDigits` to `sumDigitsWith` in practice, one would probably not rewrite them first with `id` etc., but just look at them and come up with `sumDigitsWith` directly.

Exercise 10

Write a (recursive) function `fixEq` so that `fixEq f x` repeatedly applies `f` to `x` until the result of `f` is the same as its argument.

Exercise 11

Use this function and `sumDigits` to write a function `isMultipleOf3` so that `isMultipleOf3 x` is true if repeatedly applying `sumDigits` to `x` results in 3, 6 or 9.

1.7 Anonymous functions

We defined a function `always1`, but it seems a bit silly to give a name to such a specialized and small concept. Therefore, Haskell allows us to define *anonymous functions* on the fly. The syntax is a backslash, followed by the parameter (or parameters), followed by the body of the function. So we can define `countDigits` and `sumDigits` without any helper functions like this:

```
Prelude> countDigits n = sumDigitsWith (\d -> 1) n
Prelude> sumDigits n = sumDigitsWith (\d -> d) n
```

[10] https://www.voxxed.com/2016/04/gang-four-patterns-functional-light-part-1/

These are also called *lambda abstractions*, because they are derived from the Lambda calculus, and the backslash is a poor imitation of the Greek letter lambda (`λ`).

1.8 Higher-order function definition

Lets look at the previous two definitions, and remember that when we define a function this way, we define what to replace the left-hand side with. But notice that the argument `n` is not touched at all by this definition! So we should get the same result if we simply omit it from the equation, right? And indeed, we can just as well write

```
Prelude> countDigits = sumDigitsWith (\d -> 1)
Prelude> sumDigits = sumDigitsWith (\d -> d)
```

It looks as if we just saved two characters. But what really just happened is that we shifted our perspective, and raised the level of abstraction by one layer. Instead of defining a `countDigits` as a function that takes a number and produces another number, we have defined `countDigits` as the result of instantiating the pattern `sumDigitsWith` with the function `(\d -> 1)`. At this level of thought, we do not care about the argument to `countDigits`, i.e. what it is called or so.

Exercise 12

Which other recent definitions can be changed accordingly?

1.9 Currying

We have already seen functions that *receive* a function as an argument. The way we use `twice` or `sumDigitsWith` here, we can think of them as functions that *return* functions. And this brings us to the deep and beautiful explanation why we write multiple arguments to functions the way we do: Because really, every function only ever has one argument.

We can *think* of `twice` has having two arguments (the function `f`, and the value `x`), but really, `twice` is a function that takes one argument (the function `f`), and returns another function, which then takes the value `x`. This "other" function is what we named in the above definition of `sumSumDigits`.

1.10 The composition operator ☆

Because writing code that passes functions around and modifies them (like in `twice` or `sumDigitsWith`) is so important in this style of programming, I should at this point introduce the composition operator. It is already pre-defined, but we can define it ourselves:

```
(f . g) x = f (g x)
```

The dot is a poor approximation of the mathematical symbol for function composition, "∘", and can be read as "`f` after `g`". Note `x` is passed to `g` first, and then the result to `f`.

It looks like a pretty vacuous definition, but it is very useful in writing high-level code. For example, it allows us the following, nicely abstract definition of `twice`:

```
twice f = f . f
```

Do you remember the example we used when introducing the dollar operator? We started with

```
f5 (f4 (f3 (f2 (f1 42))))
```

and rewrote it to

```
f5 $ f4 $ f3 $ f2 $ f1 42
```

Now imagine we want to abstract over 42:

```
many_fs x = f5 (f4 (f3 (f2 (f1 x))))
```

This function really is just the composition of a bunch of functions. So an idiomatic way of writing it would be

```
many_fs  = f5 . f4 . f3 . f2 . f1
```

where again, the actual value is no longer the emphasis, but rather the functions.

The value `x` is sometimes called the point (as in geometry), and this style of programming is called *point-free* (or sometimes *pointless*).

1.11 Purity

We have seen most important fundamental concepts of functional programming here. So let me point out a few things that we have not seen, and not due to lack of time, because they are not there:

We have not seen variables that you declare to hold one value, and later you update them to another value. We have not seen how to get a variable that has a random value, or one that the user has input. We have not seen ways of deleting files or launching missiles.

This is because, fundamentally, those things do not exist in Haskell. A Haskell expression simply denotes a value – e.g. a number, a Boolean, maybe a function. And it always denotes the same value. Evaluating the same expression a second time will not give different results, nor will it delete your backups. We say that Haskell is a *pure* language, and it has no *side-effects*.

Granted, there are some Haskell expressions do not denote a value: Some go into an infinite loop, or raise an exception (e.g. division by zero). But there are still no side-effects here.

Because functions a simply abstracted expressions, they are also pure: The return value depends *only* on the value of the arguments to the function; not on the time of day, the user's mood or the system's random number generator. In that sense they behave just like mathematical functions.

But if expressions and functions can't *do* anything, how can we write useful programs? Programs that respond to network requests, or do an in-place array sort, or use concurrency? Can we do that, and still have a pure language? Yes, we can, and Haskell's solution to this dilemma are monads. We will handle that topic in a quick-and-dirty way in the chapter on imperative code and more properly the chapter on monads.

1.12 Laziness ☆

As a final bit in this section, let's talk about laziness. Most often this can be ignored when reading Haskell code, and in general laziness is not as important (or as bad) as some people say it is. But it plays an important role in Haskell's support for abstraction, so let's briefly look at it.

Laziness means that an expression is evaluated as late as possible, i.e. when it is needed to make a branching decision, or when it is to be printed on the screen. We can only observe when things are being evaluated when we have side-effects, and the only side effects we can produce so far are non-termination and exceptions. So let us use division by zero to observe that the first argument to `const` is used, but the second one is not:

```
Prelude> 0 `div` 0
*** Exception: divide by zero
Prelude> const (0 `div` 0) 1
*** Exception: divide by zero
Prelude> const 1 (0 `div` 0)
1
```

To see why this is so crucial for abstraction, consider the following two functions that implement a safe version of `div` and `mod` that just returns 0 if the user tries to divide by 0:

```
Prelude> x `safeDiv` y = if y == 0 then 0 else x `div` y
Prelude> x `safeMod` y = if y == 0 then 0 else x `mod` y
Prelude> 10 `safeDiv` 5
2
Prelude> 10 `safeDiv` 0
0
```

Clearly, the definitions of `safeDiv` and `safeMod` share a pattern. So let us extract the pattern "if y is zero then return zero else do something else":

```
Prelude> unlessZero y z = if y == 0 then 0 else z
Prelude> x `safeDiv` y = unlessZero y (x `div` y)
Prelude> x `safeMod` y = unlessZero y (x `mod` y)
Prelude> 10 `safeDiv` 5
2
Prelude> 10 `safeDiv` 0
0
```

So that works. But it only works because Haskell is lazy. Consider what would happen in a strict language, where expressions are evaluated before passed to a function:

```
  10 `safeDiv` 0
= unlessZero 0 (10 `div` 0)
= unlessZero 0 (*** Exception: divide by zero
```

Only with laziness can we easily abstract not only over computations, but even over control flow, and create our own control flow constructs – simply as higher-order functions!

In fact, even `if … then … else` could just be a normal function with three parameters, defined somewhere in the standard library. The fact that there is special syntax for it is pure convenience – which, again, is not the case in strict languages.

2 Types

In the first chapter, we have seen how functional programming opens the way to abstraction, and to condense independent concerns into separate pieces of code. This is a very powerful tool for modularity, and helps to focus on the relevant part of a problem while keeping the bookkeeping out of sight. But powerful is also dangerous – using a higher order function correctly without any aid can be mind-bending.

Whenever we write functions like in the previous section, we have an idea in our head about what their arguments are – are they just numbers, or are they functions, and what kind of functions – and what do they expect and what do they return. It is obvious to us that writing `twice isEven` does not make sense, because `isEven` returns `True` or `False`, but expects a number, so it cannot be applied twice in a row.

This is all simple and obvious, but it is a lot to keep in your head as the code grows larger, and even more so once the code is changing and there are more people working on it. So to keep this power and complexity manageable, Haskell has a strong static type system, which is essentially a way for you to communicate with the compiler about these ideas you have in your head. You can ask the compiler "what do you know about this function? what can it take, what kind of things does it return?". And you can tell the compiler "this function ought to take this and return that (and please tell me if you disagree)".

In fact, many Haskellers prefer to do type-driven development: First think about and write down the type of the function they need to create, and *then* think about implementing them.

Besides communicating with the compiler, types are also crucial in communicating with your fellow developers and/or users of your API. For many functions, the type alone, or the type and the name, is sufficient to tell you what it does.

2.1 Tooling interlude: Editing files ☆

At this point, we should switch from working exclusively in the REPL to writing an actual Haskell file. We can start by creating a file `types.hs`[11], and putting in the code from the first chapter:

```
isRound x = x `mod` 10 == 0
hasLastDigit x y = x `mod` 10 == y
isHalfRound x = x `hasLastDigit` 0 || x `hasLastDigit` 5
x `divides` y = x `div` y == 0
twice f x = f (f x)
countCountDigits x = twice countDigits x
sumSumDigits x = twice sumDigits x
sumDigitsWith f n = if n < 10 then f n else sumDigitsWith f (n `div` 10) + f (n `mod` 10)
countDigits = sumDigitsWith (\d -> 1)
sumDigits = sumDigitsWith (\d -> d)
fixEq f x = if x == f x then x else fixEq f (f x)
isMultipleOf3 x = fixEq sumDigits x == 3 || fixEq sumDigits x == 6 || fixEq sumDigits x == 9
```

We can load this file into `ghci` by either starting it with `ghci Types.hs` or by typing `:load Types.hs` within `ghci`. After you change and save the file, you can reload with `:reload` (or simply `:r`).

[11] https://haskell-for-readers.nomeata.de/files/types.hs

2.2 Basic types

As I mentioned before, you can chat with the compiler about the types of things, and ask what it thinks they are. We can do that with the `:type` (or `:t`) command:

```
*Main> :t sumDigits
sumDigits :: Integer -> Integer
```

Here, GHC tells us the type of `sumDigits`, in the form of a *type annotation*, i.e. a term (`sumDigits`) followed by two colons, followed by its type. The type itself tells us that `sumDigits` is a function (as indicated by the arrow) that takes an `Integer` as an argument and returns an `Integer` as a result. Which greatly matches our expectation!

Instead of asking GHC for the type, we can also specify it, simply by adding the line

```
sumDigits :: Integer -> Integer
```

to the file (commonly directly above the function definition, rarely all bundled up in the beginning).

If we insert a type annotation that does not match what we wrote in the code, for example, if we added

```
isRound :: Integer -> Integer
```

we would get an error message like

```
types.hs:2:13: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘Bool’
    • In the expression: x `mod` 10 == 0
      In an equation for ‘isRound’: isRound x = x `mod` 10 == 0
  |
2 | isRound x = x `mod` 10 == 0
  |             ^^^^^^^^^^^^^^
```

Note that the compiler believes the type signature, and complains about the code, not the other way around.

The error message mentions a type `Bool` which, as you can guess, is the type of Boolean expressions, e.g. `True` and `False`. With this knowledge, we can write the correct type signature for `isRound`:

```
isRound :: Integer -> Bool
```

2.3 Polymorphism

Let us consider `twice` for a moment, and think about what to expect from its type. It is a function that takes two arguments, and the first argument ought to be a function itself... and here is how GHC writes this:

```
*Main> :t twice
twice :: (t -> t) -> t -> t
```

From this example, we learn that

- the type of functions with multiple arguments is written using multiple arrows,

- the function arrow can just as well occur inside an argument, namely when an argument itself is a function.

But what is this type `t`? There is not, actually, a type called `t`. Instead, this is a *type variable*, meaning that the function `twice` can be used with any type. Any lower-case identifier in a type is a type variable (not just `t`), and concrete types are always upper-case.

Here we can see that we can use `twice` with numbers, Booleans, and even with functions:

```
*Main> twice countDigits (99^99)
3
*Main> twice not True
True
*Main> twice twice countDigits (99^99)
1
```

What does not work is passing a function to `twice` that works on numbers, but then pass a `Bool`:

```
*Main> twice countDigits True

<interactive>:12:19: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘Bool’
    • In the second argument of ‘twice’, namely ‘True’
      In the expression: twice countDigits True
      In an equation for ‘it’: it = twice countDigits True
```

In other words: The `t`s in the type of `twice` can become *any* type, but it has the be the same type everywhere.

Exercise 13

What do you think is the type of `id`?

If we ask for the type of the function `const`, we see two different type variables:

```
*Main> :t const
const :: a -> b -> a
```

And here we can indeed instantiate them at different types:

```
*Main> :t const True 1
const True 1 :: Bool
```

2.4 Constrained types (a first glimpse)

There are more polymorphic functions in our initial set, for example `fixEq`:

```
*Main> :t fixEq
fixEq :: Eq t => (t -> t) -> t -> t
```

The part after the `=>` is what we expect: two arguments, the first a function, all the same types, just like with `twice`. The part before the `=>` is new: It is a *constraint*, and it limits which types `t` can be instantiated with. Remember that `fixEq` uses `(==)` to check if the value has stabilized. But not all values can be compared for equality! (In particular, functions cannot). So `fixEq` does not work with any type, but only

those that support equality. This is what `Eq t` indicates, and indeed we get an error message when we try to do it wrongly:

```
*Main> fixEq twice not True

<interactive>:27:1: error:
    • No instance for (Eq (Bool -> Bool)) arising from a use of ‘fixEq’
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: fixEq twice not True
      In an equation for ‘it’: it = fixEq twice not True
```

This `Eq` thing is not some built-in magic, but rather a *type class*, another very powerful and important feature of Haskell, which we will dive into separately later.

Other functions in our list are polymorphic where we may not have expected it:

```
*Main> :t sumDigitsWith
sumDigitsWith :: (Integral a1, Num a2) => (a1 -> a2) -> a1 -> a2
```

This is not the type we might have expected! This is because Haskell supports different numeric types, and uses type classes to overload the numeric operations. But remember that typing is a conversation: We can simply tell GHC that we want a different (more specific) type for `sumDigitsWith`, by adding

```
sumDigitsWith :: (Integer -> Integer) -> Integer -> Integer
```

to our file. In fact, in practice one *always* writes full type signatures for all top-level definitions, so this should be less of a problem for Haskell readers.

2.5 Parametricity

One obvious use for such polymorphism is to write code once, and use it at different types. But there is another great advantage of polymorphic functions, even if we only ever intend to instantiate the type variables with the same type, and that is reasoning by *parametricity*.

In a function with a polymorphic type like `twice` there is not a lot we can do with the parameters. Sure, we can apply `f` to `x`, and maybe apply `f` more than once. But that is just about all we can do: Because `x` can have an arbitrary type, we cannot do arithmetic with it, we cannot print it, we cannot even compare it to other values of type `x`.

This severely restricts what `twice` can do at all ... but on the other hand means that just from looking at the type signature of `twice` we already know a lot about what it does.

A very simple example for that is the function `id`, with type `a -> a`. *Any* function of this type will either

- return its argument (i.e. be the identity function),
- return never (i.e. go into an infinite loop), or
- raise an exception.

Exercise 14

A great example for the power of polymorphism is the following type signature:

```
(a -> b -> c) -> b -> a -> c
```

There is a function of that type in the standard library. Can you tell what it does? Can you guess its name? You can use a type-based search engine like Hoogle[12] or Hayoo[13] to find the function.

2.6 Algebraic data types

The function type is very expressive, and one can model many data structures purely with functions. But of course it is more convenient to use dedicated data structures. There are a number of data structure types that come with the standard library, in particular tuples, lists, the `Maybe` type. But it is more instructive to first look at how we can define our own.

We can declare new data types using the `data` keyword, the name of the type, the `=` sign, and then a list of *constructors*, separated by pipes (`|`). These declarations are a bit odd, since they use `=` although it is not really an equality, but let us look at it step by step.

2.6.1 Enumerations

In the simplest case, we can use this to declare an enumeration type:

```
data Suit = Diamonds | Clubs | Hearts | Spades
```

From now on, we can use the constructors, e.g. `Diamonds` as values of type `Suit`. This is how we *create* values of type `Suit` – and it is the only way, so we know that every value of type `Suit` is, indeed, one of these four constructors.

Note that `Suit` is a *type*, i.e. something you can use in type signatures, while `Diamonds` and the other constructors are *values*, i.e. something you can use in your function definitions.

When we have a value of type `Suit` then the only thing we can really do with it is to find out which of these four constructors it actually is. The way to do that is using *pattern matching*, for example using the `case … of …` syntax:

```
isRed :: Suit -> Bool
isRed s = case s of
  Diamonds -> True
  Hearts -> True
  _ -> False
```

The expression `case e of …` evaluates the *scrutinee* `e`, and then sequentially goes through the list of cases. If the value of the scrutinee matches the *pattern* left of the arrow, the whole expression evaluates to the right-hand side. We see two kinds of patterns here: Constructor patterns like `Diamonds`, which match simply when the value is, indeed, the constructor, and the *wildcard pattern*, written as an underscore, which matches any value.

It is common to immediately pattern match on the parameter of a function, so Haskell supports pattern-matching directly in the function definition:

```
isRed :: Suit -> Bool
isRed Diamonds = True
```

[12] https://www.haskell.org/hoogle/?hoogle=%28a+-%3E+b+-%3E+c%29+-%3E+b+-%3E+a+-%3E+c
[13] http://hayoo.fh-wedel.de/?query=%28a+-%3E+b+-%3E+c%29+-%3E+b+-%3E+a+-%3E+c

```
isRed Hearts = True
isRed _ = False
```

The type `Bool` that we have already used before is merely one of these enumeration types, defined as

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

and we could do without `if … then … else …` by pattern-matching on `Bool`:

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True x y = x
ifThenElse False x y = y
```

The only reason to have `if … then … else …` is that it is a bit more readable.

2.6.2 Product types

So far, the constructors were just plain values. But we can also turn them into "containers" of sort, where we can store other values. As an basic example, maybe we want to introduce a type for complex numbers:

```
data Complex = C Integer Integer
```

(Mathematically educated readers please excuse the use of `Integers` here.)

This creates a new type `Complex`, with a constructor `C`. But `C` itself is not a value of type `Complex`, but rather it is a function that creates values of type `Complex` and, crucially, it is the only way of creating values of type `Complex`. We can ask for the type of `C` and see that it is indeed just a function:

```
Prelude> :t C
C :: Integer -> Integer -> Complex
```

so it should be clear how to use it:

```
origin :: Complex
origin = C 0 0
```

Note that in the above data type declaration, `Complex` is a type, `C` is a term, but `Integer` is again a type.

Again the way to use a complex number is by pattern matching. This time we use pattern matching not to distinguish different cases – *every* `Complex` is a `C` – but to extract the parameters of the constructor:

```
addC :: Complex -> Complex -> Complex
addC (C x1 y1) (C x2 y2) = C (x1 + x2) (y1 + y2)
```

The parameter in a pattern – the `x1` here – can itself be a pattern, for example `0` (which matches only the number 0), or underscore:

```
isReal :: Complex -> Bool
isReal (C _ 0) = True
isReal _ = False
```

2.6.3 Sum types

A type like `Complex`, with exactly one constructor, is called a *product type*. But we can of course have types with more than one constructor and constructor arguments:

```
data Riemann = Complex Complex | Infinity
```

This declares a new type `Riemann` that can be built using one of these two constructors:

1. The constructor `Complex`, which takes one argument, of type `Complex`. Types and terms (including constructors) have different namespaces, so we can have a type called `Complex`, and a constructor called `Complex`, and they can be completely unrelated. This can be confusing, but is rather idiomatic.

   The type of `Complex` shows that we can use it as a function, to create a point of the `Riemann` sphere from a complex number:

   ```
   Prelude> :t Complex
   Complex :: Complex -> Riemann
   ```

2. The constructor `Infinity` takes no arguments, and simply is a value of type `Riemann` itself.

When we pattern match on a value of type `Riemann`, we learn whether it was created using `Complex` or `Infinity`, and in the former case, we also get the complex number passed to it:

```
addR :: Riemann -> Riemann -> Riemann
addR (Complex c1) (Complex c2) = Complex (c1 `add` c2)
addR Infinity _  = Infinity
addR _ Infinity  = Infinity
```

A data type that has more than one constructor is commonly called a *sum type*. Because `data` allows you to build types from sums and products, these types are called *algebraic data types* (ADTs).

2.6.4 Recursive data types

It it worth pointing out that it is completely fine to have a constructor argument of the type that we are currently defining. This way, we obtain a *recursive data type*, and this is the foundation for many important data structures, in particular lists and trees of various sorts. Here is a simple example, a binary tree with numbers on all internal nodes:

```
data Tree = Leaf | Node Integer Tree Tree
```

Again, this can be read as "a value of type `Tree` is either a `Leaf`, or it is a `Node` that contains a value of type `Integer` and references to two subtrees."

There is nothing particularly interesting about constructing such a tree (use `Leaf` and `Node`) and traversing it (use pattern matching). Here is a piece of idiomatic code that inserts a new number into a tree:

```
insert :: Integer -> Tree -> Tree
insert x Leaf = Node x Leaf Leaf
insert x (Node y t1 t2)
    | y < x     = Node y t1 (insert x t2)
    | otherwise = Node y (insert x t1) t2
```

This code shows a new, syntactic feature: Pattern guards! These are Boolean expressions that you can use to further restrict when a case is taken. The third case in this function definition is only used if `y < x`, otherwise the following cases are tried. Of course this could be written using `if … then … else …`, but the readability and aesthetics are better with pattern guards. The value `otherwise` is simply `True`, but this reads better.

Exercise 15

Consider the following definition:

```
data Wat = Wat Wat
```

Is this legal? What does it mean? Which occurrences of `Wat` are terms, and which are types? Can you define a value of type `Wat`?

2.6.5 Polymorphic data types

The tree data type declared in the previous section ought to be useful not just for integers, but maybe for any type. But it would be seriously annoying to have to create a new tree data type for each type we want to store in the tree. Therefore, we have *polymorphic data types*. In the example of the tree, we can write:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

You may remember that the a here ought to be a type variable, because it occurs in the place of a type, but is lower-case. When we want to use this tree data type at a concrete type, say `Integer`, we simply write `Tree Integer`:

```
insert :: Integer -> Tree Integer -> Tree Integer
```

The actual code of the function does not change.

We can also write functions that work on polymorphic trees, i.e. which we can use on *any* `Tree`, no matter what type the values in the nodes are. A good example is:

```
size :: Tree a -> Integer
size Leaf = 0
size (Node _ t1 t2) = 1 + size t1 + size t2
```

Again, parametricity makes the type signature of such a function more useful than it seems at first: Just from looking at the type signature of `size` we *know* that this function does not look at the values stored in the nodes. Together with the name, that is really all the documentation we might need. Compare this to `size :: Tree Integer -> Integer` – now it could just as well be that this function includes the number stored in the node in the result somehow.

2.6.6 Functions in data types ⋆

Maybe this is obvious to you, after the emphasis on functions in the first chapter, but it is still worth pointing out that data type can also store functions. This blurs the distinction between data and code some more, as this nice example shows:

```
data Stream a b
    = NeedInput (a -> Stream a b)
```

```
    | HasOutput b (Stream a b)
    | Done
```

The type `Stream a b` models a state machine that consumes values of type `a`, produces values of type `b`, and maybe eventually stops. Such a machine is in one of three states:

1. Waiting for input. This uses the `NeedInput` constructor, which carries a *function* that consumes a type of value `a` and returns the new state of the machine.
2. Producing output. This uses the `HasOutput` constructor, which stores the output value of type `b`, and the subsequent state of the machine.
3. `Done`.

We can create a state machine that does run-length encoding this way. This one does not ever stop, but that's fine:

```
rle :: Eq a => Stream a (Integer,a)
rle = NeedInput rle_start

rle_start :: Eq a => a -> Stream a (Integer, a)
rle_start x = NeedInput (rle_count x 1)

rle_count :: Eq a => a -> Integer -> a -> Stream a (Integer, a)
rle_count x n x' | x == x' = NeedInput (rle_count x (n + 1))
                 | otherwise = HasOutput (n, x) (rle_start x')
```

2.7 Predefined data types

We intentionally discussed the mechanisms of algebraic data types first, so that we can explain the most common data types in the standard library easily.

2.7.1 Booleans ☆

As mentioned before, the values `True` and `False` are simply the constructors of a data type defined as

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

There is nothing magic about the definition of `Bool`. But this type plays a special role because of pattern guards and the `if … then … else …` construct.

2.7.2 `Maybe`

A very common use case for algebraic data types is to capture the idea of a type whose values "maybe contain nothing, or just a value of type `a`". Because this is so common, such a data type is predefined:

```
data Maybe a = Nothing | Just a
```

You might see `Maybe`, for example, in the return type of a function that deserializes a binary or textual representation of a type, for example:

```
parseFoo :: String -> Maybe Foo
```

Such an operation can fail, and if the input is invalid, it would return `Nothing`. As a user of such a function, the only way to get to the value of type `a` therein is to pattern-match on the result, which forces you to think about and handle the case where the result is `Nothing`.

This is much more robust than the common idiom in C, where you have to remember to check for particular error values (-1, or `NULL`), or Go, where you get both a result and a separate error code, but you can still be lazy and use the result without checking the error code.

A big part of Haskell's reputation as a language that makes it easier to write correct code relies on the use of data types to precisely describe the values you are dealing with.

Exercise 16

How many values are there of type `Maybe (Maybe Bool)`. When can it be useful to nest `Maybe` in that way?

2.7.3 `Either` ⋆

With maybe we can express "one or none". Sometimes we want "one or another" type. For this, the standard library provides

```
data Either a b = Left a | Right b
```

Commonly, this type is used for computations that can fail, but that provide some useful error messages when they fail:

```
parseFoo :: String -> Either ParseError Foo
```

This gives us the same robustness benefits of `Maybe`, but also a more helpful error messages. If used in this way, then the `Left` value is always used for the error or failure case, and the `Right` value for when everything went all right.

2.7.4 Tuples

Imagine you are writing a function that wants to return two numbers – say, the last digit and the rest of the number. The way to do that that you know so far would require defining a data type:

```
data TwoIntegers = TwoIntegers Integer Integer
splitLastDigit :: Integer -> TwoIntegers
splitLastDigit n = TwoIntegers (n `div` 10) (n `mod` 10)
```

Clearly, the concept of "passing around two values together" is not particularly tied to `Integer`, and we can use polymorphism to generalize this definition:

```
data Two a b = Two a b
splitLastDigit :: Integer -> Two Integer Integer
splitLastDigit n = Two (n `div` 10) (n `mod` 10)
```

And because this is so useful, Haskell comes with built-in support for such pairs, including a nice and slim syntax:

```
data (a,b) = (a,b)
splitLastDigit :: Integer -> (Integer, Integer)
splitLastDigit n = (n `div` 10, n `mod` 10)
```

Besides tuples, which store two values, there are triples, quadruples and, in general, n-tuples of any size you might encounter. But really, if these tuples get larger than two or three, the code starts to smell.

Their size is always fixed and statically known at compile time, and you can have values of different types as components of the tuple. This distinguishes them from the lists we will see shortly.

Exercise 17

How could you represent the Riemann numbers from the previous section using only these predefined data types?

Useful predefined functions related to tuples are

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

and because of their type I do not have to tell you what they do.

2.7.5 The unit type

There is also a zero-tuple, so to say: The unit type written `()` with only the value `()`:

```
data () = ()
```

While this does not look very useful yet, we will see that it plays a crucial role later. Until then, you can think of it as a good choice when we have something polymorphic, but we do not actually need an "interesting" type there.

Exercise 18

Write functions

```
fromEitherUnit :: Either () a -> Maybe a
```

and

```
toEitherUnit :: Maybe a -> Either () a
```

that are inverses to each other.

In only one of these type signatures you can replace `()` with a new type variable `b`, and still implement the function. In which one? Why?

2.7.6 Lists

In the previous section we defined trees using a recursive data type. It should be obvious that we can define lists in a very analogous way:

```
data List a = Empty | Link a (List a)
```

This data structure is so ubiquitous in functional programming that it not only comes with the standard library, it also has very special, magic syntax:

```
data [a] = [] | a : [a]
```

In words: The type `[a]` is the type of lists with values of type `a`. Such a list is either the empty list, written as `[]`, or it is a non-empty list containing of a head `x` of type `a`, and a tail `xs`, and is written as `x:xs`. Note that the constructor `(:)`, called "cons", is using operator syntax.

There is even more special syntax for lists:

1. Finite lists can be written as `[1, 2, 3]` instead of `1 : 2 : 3 : []`.
2. Lists of numbers can be enumerated, e.g. `[1..10]`, or `[0,2..10]`, or even (due to laziness) `[1..]`.
3. List comprehensions look like `[ (x,y) | x <- xs, y <- ys, x < y ]`, reminiscent of the set comprehension syntax from mathematics. We will not discuss them now, I just wanted to show them and give you terms to search for.

Common operations on lists worth knowing are `(++)` to concatenate two lists, and `map` to apply a function to each element of list:

```
(++) :: [a] -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

Lists are very useful for many applications, but they are not a particularly high-performance data structure – random access and concatenation is expensive, and they use quite a bit of memory. Depending on the application, other types like arrays/vectors, finger trees, difference lists, maps or sets might be more suitable.

2.7.7 Characters and strings ⋆

Unexpectedly, Haskell has built-in support for characters and text. A single character has type `Char`, and is written in single quotes, e.g. `'a'`, `'☃'`, `'\''`, `'\0'`, `'\xcafe'`. These character are Unicode code points, and not just 7 or 8 bit characters.

The built-in type `String` is just an alias for `[Char]`, i.e. a list of characters. Haskell supports special built-in syntax for strings, using double quotes, but this is just syntactic sugar to the list syntax:

```
Prelude> "hello"
"hello"
Prelude> ['h','e','l','l','o']
"hello"
Prelude> 'h':'e':'l':'l':'o':[]
"hello"
```

Because `String` is built on the list type, all the usual list operations, in particular `(++)` for concatenation, work on strings as well.

But `String` also has the same performance issues as lists: While it is fine to use them in non-critical parts of the code (diagnostic and error messages, command line and configuration file parsing, filenames), `String` is usually the wrong choice if large amounts of strings need to be processed, e.g. in a templating library. Additionally libraries provide more suitable data structures, in particular `ByteString` for binary data and `Text` for human-readable text.

2.8 Records ☆

Assume you want to create a type that represents an employee in a HR database. There are a fair number of field to store – name, date of birth, employee number, room, login handle, public key etc. You could use a tuple with many fields, or create your own data type with a constructor with many fields, but either way you will have to address the various fields by their position, which is verbose, easy to get wrong, and hard to extend.

In such a case, you can use records. These allow you to give names to the *field* of a constructor, and get some convenience functions along the way. Here we see how to declare and use them

```
data Employee = Employee
    { name :: String
    , room :: Integer
    , pubkey :: ByteString
    }

theBoss :: Employee
theBoss = Employee { name = "Don Vito Corleone", room = 101, pubkey = "0xAD…" }

willDrownWhenTheFloodComes :: Employee -> Bool
willDrownWhenTheFloodComes Employee { room = r } = r < 200

moveOneLevelUp :: Employee -> Employee
moveOneLevelUp e = e { room = new_room }
  where new_room = room e + 100
```

Record syntax has five aspects:

1. The declaration has special syntax. In terms of the constructor `Employee`, the declaration is equivalent to

   ```
   data Employee = Employee String Integer ByteString
   ```

   and it is always possible to use `Employee` as a normal prefix function in terms and patterns. But the record syntax declaration enables the following nice syntactic devices:

2. Record creation: Instead of `Employee n r p` you can write `Employee { name = n; pubkey p= p; room = r }`, and the order of the fields becomes conveniently irrelevant.

3. Record pattern matching. You can also write `Employee { name = n; room = r; pubkey = p }` in a pattern, to match on `Employee` and get `n`, `r` and `p` into scope.

4. Record update syntax: If we have `e :: Employee`, then `e { room = r' }` is like an `Employee` and all fields are the same as `e` with the exception of `room`.

5. The names of the fields are available as getters, i.e. after the above definition of `Employee`, there is a function `room :: Employee -> Integer` etc.

Curiously, the record creation or update syntax binds closer than function applications: `g x { f = y }` is `g (x { f = y })`, and *not* `(g x) { f = y }`.

With the language extension `RecordWildCards` enabled, it is even possible to write `Employee{..}` in a

pattern, and get *all* fields of the employee record into scope, as variables, as if one had written `Employee { name = name; room = room; pubkey = pubkey }` (although some say that's bad style, because it is too implicit).

2.9 Newtypes ☆

Sometimes you will see a type declaration that uses `newtype` instead of `data`:

```
newtype Riemann = Riemann (Maybe (Integer, Integer))
```

For all purposes relevant to us so far you can mentally replace `newtype` with `data`. There are difference in memory representation (a `newtype` is "free" in some sense), but that is, at this level, irrelevant for us.

2.10 Type synonyms ☆

Haskell allows you to introduce new names for existing types. One example is the type `String`, which is defined as

```
type String = [Char]
```

With this declaration, you can use `String` instead of `[Char]` in your type signatures. They are completely interchangeable, and a value of type `String` is still just a list of characters.

So type synonyms do not introduce any kind of type safety, they merely make types more readable.

2.11 Haddock ☆

Because knowing the type of a function is already a big step towards understanding what it does, the usual way of documenting a Haskell API is very much centered around types. The tool `haddock` creates HTML pages from Haskell source files that list all functions with their type, and – if present – the documentation that is attached to it via a comment.

For Haskell libraries hosted in the central package repository *Hackage*, this documentation is also provided. For example, you can learn all about the types and functions that are available by default by reading the haddock page for the prelude[14]. (There is a bunch of noise there that might not be relevant to you, like long lists of "Instances". You can skip over them.)

From this documentation you will also find links labeled "Source" that take you to the definition of a type or function in the source code, in a syntax-highlighted and crosslinked presentation of the source.

Relatedly, you can also effectively search for functions with a certain type, using type-based search engine like Hoogle[15] or Hayoo[16]. These can also be set-up in house to index your private code base.

[14] https://hackage.haskell.org/package/base/docs/Prelude.html
[15] https://www.haskell.org/hoogle/
[16] http://hayoo.fh-wedel.de/

3 Code structure small and large

The next big topic we need to learn is how programmers structure their code. This happens on multiple levels

- in a function: intermediate results are named, local helper functions are defined.
- within a file: functions, type signatures, and documentation fragments are arranged.
- within a project (library, package): code is spread out in different files, and imported from other files.
- between projects: packages are versioned, equipped with meta-data, and depend on each other.

3.1 `let`-expressions

The most basic way of adding some structure within an expression is to give a name to a subexpression, and possibly use it later. So instead of

```
isMultipleOf3 x = fixEq sumDigits x == 3 || fixEq sumDigits x == 6 || fixEq sumDigits x == 9
```

one could write

```
isMultipleOf3 x =
  let y = fixEq sumDigits x
  in y == 3 || y == 6 || y == 9
```

which is arguably easier to read.

Be careful: A `let` expression in Haskell can always be recursive, so

```
isMultipleOf3 x =
  let x = fixEq sumDigits x
  in x == 3 || x == 6 || x == 9
```

might not do what you expect.

In such a `let` expression, you can also do pattern-matching, e.g. to unpack a tuple:

```
sumDigitsWith :: (Integer -> Integer) -> Integer -> Integer
sumDigitsWith f n
  | n < 10 = f n
  | otherwise =
    let (r,d) = splitLastDigit n
    in sumDigitsWith f r + f d
```

This is a fine and innocent thing to do if the pattern is *irrefutable*, i.e. always succeeds, but is a code smell if it is a pattern that can fail, e.g. `let Just x = something in ...`. In the latter case, a `case` statement might be more appropriate.

We can also define whole functions in a `let`-expression, just like on the top level. This might improve the code of our run-length-encoding automaton:

```
rle :: Eq a => Stream a (Integer,a)
rle =
  let start x = NeedInput (count x 1)
```

```
      count x n x' | x == x' = NeedInput (count x (n + 1))
                   | otherwise = HasOutput (n, x) (start x')
  in NeedInput start
```

One advantage of this is that the "internal" functions `start` and `count` are now no longer available from the outside, and so a reader of this code knows for sure that these are purely internal. We can also drop the `rle_` prefix.

Another important advantage is that such local functions have access to the parameters of the enclosing function. To see this in action, let us extend `rle` with a parameter that indicates an element of the stream that should make the automaton stop:

```
rle :: Eq a => a -> Stream a (Integer,a)
rle stop =
  let start x | x == stop = Done
              | otherwise = NeedInput (count x 1)
      count x n x' | x == x' = NeedInput (count x (n + 1))
                   | otherwise = HasOutput (n, x) (start x')
  in NeedInput start
```

In `start` we can now access `stop` just fine. If `start` and `count` were not local functions, then we would have to add `stop` as an explicit parameter to *both* local functions, significantly cluttering the code with administrative details.

3.2 `where`-clauses ⋆

I think few syntactic features show that Haskell's syntax is designed with readability in mind, valuing that higher than syntactic minimalism, as well as the `where` clauses.

Looking the previous version of the `rle` program, a very picky reader might complain that it is annoying to have to first read past `start` and `count` to see the last line, when the last line is logically the first to be executed.

Therefore, the programmer has the option to use a `where`-clause instead of a `let` expression here. A `where`-clause is attached to a function equation (or, more rarely, to a match in a `case` expression), has access to its parameters and – most crucially – is written after or below the right-hand side of the equation:

```
rle :: Eq a => a -> Stream a (Integer,a)
rle stop = NeedInput start
  where
    start x | x == stop = Done
            | otherwise = NeedInput (count x 1)

    count x n x' | x == x' = NeedInput (count x (n + 1))
                 | otherwise = HasOutput (n, x) (start x')
```

It is not a huge change, but one that – in my humble opinion – improves readability by a small but noticeable bit.

If you have a function with multiple guards on one equation, such as `start`, then a `where` clause scopes over all such guards. So we could write

```
sumDigitsWith :: (Integer -> Integer) -> Integer -> Integer
sumDigitsWith f n
  | n < 10 = f d
  | otherwise = sumDigitsWith f r + f d
  where (r,d) = splitLastDigit n
```

(note that `d` is used in both right-hand sides) if we wanted.

3.3 Comments ☆

Of course, Haskell supports comments. There are line comments and multi-line comments:

```
answer = 42 -- but what is the question?

{-
In the following code, we write a function that correctly tells
us whether a Turing machine halts:
-}
halts :: TuringMachine -> Bool
halts turing_machine = halts turing_machine
```

The `haddock` Haskell documentation tool uses specially marked comments for documentation, so the above could better be written as (note the vertical bar):

```
{- |
In the following code, we write a function that correctly tells
us whether a turing machine halts:
-}
halts :: TuringMachine -> Bool
halts turing_machine = halts turing_machine
```

3.4 The structure of a module

As we zoom out one step, we get to look at a Haskell file as a whole. In Haskell, every file is also a Haskell *module*, and modules are used to organize namespaces.

Normally, a Haskell module named `Foo.Bar.Baz` lives in a file `Foo/Bar/Baz.hs`, and begins with

```
-- in file Foo/Bar/Baz.hs
module Foo.Bar.Baz where
```

Haskell module names are always capitalized.

If a file does not have such a header (which is usually only the case for experiments and the entry point of a Haskell program), then it is implicitly called `Main`. This is why the GHCi prompt says `Main>` after loading such a file.

The rest of the module is are declarations: Values, functions, types, type synonyms etc. The important bit to know here is that the order of declarations is completely irrelevant: You can use functions you that are defined further down, you can mix type and function declarations, you can even separate the type signature of a function from its definition (but you have to keep multiple equations of one function

together). This allows the author to sort functions by topic, or by relevance, rather than by dependency, and it is not uncommon to first show the main entry-point of a module, and put all the helper functions it uses below.

3.5 Importing other modules

Obviously, the point of having multiple files of Haskell code is to use the code from one in the other. This is achieved using `import` statements, which *must* come right after the `module` header, and before any declarations.

So if we have a file `Target.hs` with content

```
-- file Target.hs
module Target where
who :: String
who = "world"
```

and another file `Tropes.hs` with content

```
-- file Tropes.hs
module Tropes where

import Target

greeting :: String
greeting = "Hello " ++ who ++ "!"
```

then the use of `who` in `greeting` refers to the definition in the file `Target.hs`.

We could also write

```
greeting :: String
greeting = "Hello " ++ Target.who ++ "!"
```

and use the *fully qualified* name of `who`. This can be useful for disambiguation, or simply for clarity. There must not be spaces around the period, or else it would refer to the composition operator.

If we only ever intend to refer the things we import from a module by their qualified names, then we can use a *qualified* import:

```
import qualified Target
```

This does not bring any unqualified names into scope.

And if the module has a long name, we can shorten it:

```
import qualified Target as T
```

and write `T.who`. This is a common idiom for modules like `Data.Text` that export many names that would otherwise clash with names from the prelude.

The standard library, called `base`, comes with many modules you can import[17] in addition to the `Prelude` module, which is always imported implicitly.

[17] http://hackage.haskell.org/package/base

3.6 Import lists ⋆

If we do not want to import *all* names of another module, we can import just a specific selection, e.g.:

```
import Data.Maybe (mapMaybe)
```

This makes it easier for someone reading the code to locate where a certain function is from, and it makes the code more robust against breakage when a new version of the other module starts exporting additional names. These would not silently override other names, but cause compiler errors about ambiguous names.

When including an operator in this list, include it in parentheses:

```
import Data.Function ((&), on)
```

You can import types just as well, just include them in the list. To import *constructors* (which look like types), you have to list them after the type they belong to. So if we put our definitions of `Complex` and `Riemann` into a file `Riemann.hs`, namely

```
-- file Riemann.hs
module Riemann where
data Complex = C Integer Integer
data Riemann = Complex Complex | Infinity
```

then you can import everything using

```
import Riemann (Complex(C), Riemann(Complex, Infinity))
```

or, shorter,

```
import Riemann (Complex(..), Riemann(..))
```

3.7 Export lists and abstract types ⋆

You can not only restrict what you import, but also what you export. To do so, you list the names of functions, types, etc. that you want to export after the module name:

```
module Riemann (Complex, Riemann(..)) where
```
…

A short export list is a great help when trying to understand the role and purpose of a module: If it only exports one or a small number of functions, it is clear that these are the (only) entry points to the code, and that all other declarations are purely internal, and may be refactored without affecting anything else.

By excluding the constructors of a data type from the export list, as we did in this example with the `Complex` type, we can make this type *abstract*: Users of our module now have no knowledge of the internal structure of `Complex`, and they are unable to create or arbitrarily inspect values of type `Complex`. Instead, they are only able to do so using the *other* functions that we export along with `Complex`. This way we can ensure certain invariant in our types – think of a search tree with the invariant that it is sorted – or reserve the ability to change the shape of the type without breaking depending code.

Proper user of abstract types greatly helps to make code more readable, more maintainable and more robust, quite similar to how polymorphism does it on a smaller scale.

3.8 Language extensions

Haskell is a language with a reasonably precise specification, the *Haskell Report*. When someone mentions Haskell 98, they refer to Haskell as specified in the Haskell Report from 1998[18]. There was one revision, the Haskell Report from 2010[19], with only rather small changes.

Since 1998, Haskell developers and implementors wanted to add more and more features to the language. But the report was written, and the compiler writers wanted to support Haskell, as specified, by default. Therefore, the system of *language extensions* was introduced.

A language extension is a feature that extends Haskell98 in some way. It could add more syntactic sugar, additional features on the type system or enable whole meta-programming facilities. A Haskell source file needs to explicitly declare the extensions they are using, right at the top before the `module` header, and a typical Haskell file these days might start with a number of them, and look like this:

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE CPP #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE PartialTypeSignatures #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies #-}
module Foo where
```

These language extensions (and there are many of them) are documented in the GHC user's guide[20].

You can also enable language extensions on the GHCi prompt, e.g. using `:set -XRecordWildCards`.

3.9 Haskell packages ⋆

Zooming out some more, we come across packages: A *package* is a collection of modules that are bundled under a single package name. A package contains meta-data (name, version number, author, license...). Packages declare which other packages they depend upon, together with version ranges. All this meta-data can be found in the *Cabal file* called `foo.cabal` in the root directory of the project.

Almost all publicly available Haskell packages are hosted centrally on Hackage[21], including the haddock-generated documentation and cross-linked source code. They can be easily installed using the `cabal` tool[22], or alternative systems like `stack`[23] or `nix`[24]. The packages on Hackage cover many common needs and it is expected that a serious Haskell project depends on dozen of Haskell packages from Hackage.

[18] https://www.haskell.org/onlinereport/
[19] https://www.haskell.org/onlinereport/haskell2010/
[20] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#language-options
[21] http://hackage.haskell.org/packages/
[22] https://www.haskell.org/cabal/
[23] https://www.haskellstack.org/
[24] https://nixos.org/nixpkgs/manual/#users-guide-to-the-haskell-infrastructure

4 Imperative (looking) Haskell ☆

In the first chapter I explained that Haskell is pure; its expressions simply denote values, but they do not *do* anything. Yet people out there write useful programs with Haskell. How is that possible?

The real reason is a beautiful and elegant concept called a monad, and we have a whole chapter dedicated to monads. If you plan to read that, you can skip the present chapter. But if you are eager to see how Haskell code can read user input and write to files, then this chapter provides a quick introduction to reading such imperative code.

Beware: This chapter is full of half-truths and glossing over technical details. Imagine plenty of "it looks as if" and "one can think of this as" sprinkled throughout it. Nevertheless, it is useful to get you started.

4.1 `IO`-functions

Previously we said that Haskell functions are pure functions in the mathematical sense: Given some input, they calculate some output, but nothing else can happen, and nothing besides the arguments can influence the result. This is great, but how can Haskell programs then write to files, or respond to network requests, or come up with random numbers?

The solution are `IO`-functions. These functions can be *executed*, and when such a function is executed, it can do all these nasty things, before returning a value. Here is a selection of `IO`-functions available by default:

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

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

You can see that these functions may have arguments, just as normal functions. The important bit is the return type, which is `IO Something`. This indicates that these functions can be executed, and that they have to be executed before we get our hands on the result.

Not all functions have an interesting result (e.g. `putStrLn` does not); this is where the unit type comes in handy.

4.2 The `main` function and `do` notation

To execute these functions, we have to use a special syntax, called `do`-notation, that allows us to write code in an imperative style. Here is an example:

```
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    putStrLn "Where do you want to copy it to?"
    to <- getLine
    content <- readFile from
    putStrLn ("Read " ++ show (length content) ++ " bytes.")
```

```
    writeFile to content
    putStrLn "Done copying."
```

we can compile and run this program, and it indeed copies a file:

```
$ ghc --make copy.hs
[1 of 1] Compiling Main             ( copy.hs, copy.o )
Linking copy ...
$ ./copy
Which file do you want to copy?
copy.hs
Where do you want to copy it to?
copy2.hs
Read 287 bytes.
Done copying.
$ diff copy.hs copy2.hs
```

Looking at the code, it doesn't look much different than the equivalent in a language like C or Python. Things to notice:

- The `main` function of the module is special. Just like in C, it is the entry point for a compiled Haskell program. When we run the program, then the `main` function is executed. This is the only way to start executing `IO`-functions – we cannot do that just nilly-willy within other code.
- The body of the `main` function is written as a `do` block, which clearly signposts the imperative nature of this code: It is a sequence of things to do.
- Every line below the `do` block is one execution of an `IO`-function. The first one, for example, prints a question on the terminal.
- Some of these `IO`-functions return values that we want to use later on. These we *bind* to variables, using the `<-` syntax. (The last line of a `do` block is never such a binding, can you imagine why?)
- The `main` function has type `IO ()`. So it is one of these `IO`-functions as well.

4.3 Writing `IO` functions

We do not only want to execute `IO` functions, but also define our own. This is not hard, and we have actually seen that before – the `main` function is one. We can add parameters without further problems:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile from to = do
    content <- readFile from
    putStrLn ("Read " ++ show (length content) ++ " bytes.")
    writeFile to content

main :: IO ()
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    putStrLn "Where do you want to copy it to?"
    to <- getLine
    copyFile from to
```

```
    putStrLn "Done copying."
```

All our knowledge about defining functions – parameters, pattern matching, recursion – applies here as well.

4.4 The `return` function

The last `IO` function executed in a `do` block of an `IO` function also determines its return value. Therefore we need the little function

```
return :: a -> IO a
```

if, at the end of an `IO` function, we *only* want to return something:

```
fileSize :: FilePath -> IO Integer
fileSize path = do
  content <- readFile path
  return (length content)
```

Important: Note that `return` does *not* alter the control flow. It does not make the function return. It merely specifies the return values of the current *line*.

Exercise 19

What does this program print?

```
theAnswer :: IO Integer
theAnswer = do
  putStrLn "Pondering the question..."
  return 23
  return 42

main :: IO ()
main = do
  a <- theAnswer
  putStrLn (show a)
```

4.5 Passing `IO` functions around

Just passing arguments to `copyFile` does not actually do anything: we really have to execute it, and execution happens when a function is executed from `main` (directly or indirectly). Let me demonstrate this point:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile from to = do
    content <- readFile from
    putStrLn ("Read " ++ show (length content) ++ " bytes.")
    writeFile to content

ignore :: a -> IO ()
ignore unused = putStrLn "I ignore my argument!"
```

```
main :: IO ()
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    putStrLn "Where do you want to copy it to?"
    to <- getLine
    ignore (copyFile from to)
    putStrLn "Done copying."
```

Executing this program will ask for the filenames, but it will not actually copy anything. This is because, although we passed all the required arguments to `copyFile`, we did not actually execute it.

That said, the problem was not that we passed `copyFile from to` as an argument to a function. Rather, the problem was that `ignore` did not do anything with it. We can fix that easily (and rename the function to `don'tignore` along the way):

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile from to = do
    content <- readFile from
    putStrLn ("Read " ++ show (length content) ++ " bytes.")
    writeFile to content

don'tignore :: IO () -> IO ()
don'tignore action = do
    putStrLn "About to execute the action."
    action
    putStrLn "I executed the action."

main :: IO ()
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    putStrLn "Where do you want to copy it to?"
    to <- getLine
    don'tignore (copyFile from to)
    putStrLn "Done copying."
```

This way, the `copyFile from to` function receives its parameters in the `main` function, but *is not yet executed*. It is then passed to `don'tignore`, which does something else first (it prints `"About to execute the action."`), and *then* executes the action.

```
$ ./copy
Which file do you want to copy?
copy.hs
Where do you want to copy it to?
copy2.hs
About to execute the action.
Read 549 bytes.
I executed the action.
```

```
Done copying.
```

Being able to abstract over `IO`-functions just like over anything else, and having precise control when they are *executed* (rather than just passed around), is a very powerful tool.

Exercise 20

What does this program do?

```
foo :: Integer -> IO () -> IO ()
foo 0 a = putStrLn "Done"
foo n a = do
   if n == 1 then putStrLn "Almost done"
             else return ()
   a
   foo (n-1) a

main :: IO ()
main = do
   foo 4 (putStrLn "Hooray!")
   foo 0 (putStrLn "And up she rises.")
```

4.6 `let` **in** `do` ⋆

You can use `let` expressions in `do` blocks, omitting the `in`. These work like normal `let` expressions, i.e. simply give a name to an expression:

```
main :: IO ()
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    let to = from ++ ".bak"
    copyFile from to
    putStrLn ("Created backup at " ++ to)
```

Note that it does *not* execute anything, as this example shows:

```
main :: IO ()
main = do
    putStrLn "Please press enter."
    input1 <- getLine
    putStrLn "Enter pressed."

    putStrLn "Please press enter."
    let input2 = getLine
    putStrLn "Enter pressed."
```

If we run this only the first occurrence to `getLine` actually does something:

```
$ ghc --make let-do.hs
[1 of 1] Compiling Main             ( let-do.hs, let-do.o )
```

```
Linking let-do ...
$ ./let-do
Please press enter.

Enter pressed.
Please press enter.
Enter pressed.
```

The variable `input2` is named misleadingly: It does not name any user input, the way it is define it is merely an alternative name for the `IO` function `getLine`.

4.7 The <$> operator ⋆

You will come across code that wants to execute an `IO` function *and* apply some normal (pure) function to its result in one go, like the `fileSize` function above. We can use the <$> operator to write that in one line, without giving a name to the intermediate value:

```
fileSize :: FilePath -> IO Integer
fileSize path = length <$> readFile path
```

When reading such code, you can think of <$> as a variant of `$`, with the difference that the *return value* of the expression on the right hand side is passed to the function on the left, and not the `IO` function as a whole.

5 Type classes

The language features we have seen so far can be found, with slight variations, in most functional programming languages. In this chapter, we will look at a feature that Haskell is particularly renown for: *Type classes*.

Let me start by pointing out what type classes are not: They are not classes as we know them from object oriented programming, so please do not try to attempt to understand them by analogy to that.

Instead, type classes are a language feature that provides, in sequence of sophistication,

- overloading of operators and function,
- implicit dependency injection,
- polymorphism over types with structure, and
- type-driven code synthesis.

We have actually seen most of these applications already:

5.1 Overloading

Assume, for a moment, that `(==)` operator we have seen already only works on `Integer`. Surely, it is no problem to define equality on, say, `Complex` and `Riemann`:

```
eqComplex :: Complex -> Complex -> Bool
eqComplex (C x1 y1) (C x2 y2) = x1 == x2 && y1 == y2
```

```
eqRiemann :: Riemann -> Riemann -> Bool
eqRiemann (Complex c1) (Complex c2) = c1 `eqComplex` c2
eqRiemann Infinity Infinity = True
eqRiemann _ _ = False
```

Similarly, we can define comparisons, numeric operators etc. This is good enough to express most of the code that we want to write, but it is terribly verbose and annoying to remember the name of the right equality function, and have the type (in the name) be repeated all over the code.

What we really want is to use the nice `(==)` syntax, but we want it to mean *different things at different types* – overloading!

In order to do that, we first have to declare that `(==)` is an operator name that can be overloaded, by declaring a class with it as a method:

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

Of course, the `Eq` class is already defined. From now on, I can use `(==)` with every type that is an *instance* of `Eq`. We can declare instances for `Complex` and `Riemann`:

```
instance Eq Complex where
    C x1 y1 == C x2 y2 = x1 == x2 && y1 == y2
instance Eq Riemann where
    (==) = eqRiemann
```

and with this in place, we can use `(==)` not only for `Integer`, but also `Complex` and `Riemann`. In fact, we can use `(==)` instead of `eqComplex` in the definition of `eqRiemann` – remember that the order of declarations is irrelevant in a Haskell module.

Now we can hopefully better understand the type signature of `(==)`:

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

The part after the `=>` indicates the argument and return types of `(==)`. But the type variable `a` cannot just be any type (as it was the case with fully polymorphic functions like `const`): It has to be a type that is an instance of the `Eq` class. This is expressed by the *constraint* on the left of the `=>`.

Exercise 21

Write an `Eq` instance for `Employee`, using record accessors. Is there a problem with this code?

5.2 Implicit dependency injection

Sometimes type classes are used for implicit dependency injection. Which is just a fancy way of saying "I don't want to pass an extra argument and instead want the compiler do that implicitly for me, based on the types".

Recall our definition of `fixEq`:

```
fixEq :: Eq a => (a -> a) -> a -> a
fixEq f x = if x == f x then x else fixEq f (f x)
```

which "iterates `f` on `x` until the value is equal to the one before.". Compare this to the following version:

```
fixBy :: (a -> a -> Bool) -> (a -> a) -> a -> a
fixBy p f x = if x `p` f x then x else fixBy p f (f x)
```

This function iterates until a user-specified function tells it to stop. This might be useful for some iterative approximation algorithm, where we stop once the difference between subsequent approximations is smaller than some epsilon.

Note that this is a form of dependency injection: The caller of `fixBy` passes along the dependency "stopping function". In general, this can be of course much more complex, e.g. a storage backend.

The function `fixBy` is clearly more general than `fixEq`, as we can implement `fixEq` using `fixBy`, by specifying `p` to be equality:

```
fixEq :: Eq a => (a -> a) -> a -> a
fixEq = fixBy (==)
```

So whenever a parameter to a polymorphic function is, for the given concrete type, the same, one can use type classes and overloading to make this parameter selection implicit.

Note that just because one *can* do that, one does not have to. Often explicit arguments are easier to understand and maintain and more flexible (e.g. if you might want different argument that have the same types). I consider this application of type classes less relevant than the other application presented here.

5.3 Polymorphism over types with structure

Similar to the `Eq` type class, there is an `Ord` type class that overloads the comparisons operators `(<)` `(<=)`, `(>)` and `(>=)`.

Using them, we can define a predicate on polymorphic trees that checks whether the tree is sorted, i.e. every value is less than or equal to every value further right. One way of implementing this is:

```
isSorted :: Ord a => Tree a -> Bool
isSorted = everyNode $ \y t1 t2 ->
    everyValue (\x -> x <= y) t1 &&
    everyValue (\z -> y <= z) t2

everyValue :: (a -> Bool) -> Tree a -> Bool
everyValue p = everyNode (\x _ _ -> p x)

everyNode :: (a -> Tree a -> Tree a -> Bool) -> Tree a -> Bool
everyNode p (Node x t1 t2) = p x t1 t2 && everyNode p t1 && everyNode p t2
everyNode _ Leaf = True
```

(Can you make sense of this code? It is a good exercise to make sure you can read this code. If you think this code to be inefficient, then you are right: It is algorithmically bad, but serves nicely as a high-level specification.)

Does this correctly implement the specification? Yes and no! If we have a tree of `Integer`, then `isSorted` will indeed return `True` if every element is smaller or equal to every element further on the right. But here is a counter example, involving a three-valued type:

```
data ABC = A | B | C deriving Eq
instance Ord ABC where
  x <= y | x == y = True
  A <= B = True
  B <= C = True
  _ <= _ = False
```

(You can ignore the `deriving Eq` for a moment). Now we can have a tree that is claimed to be sorted, but it is not!

```
*Main> isSorted (Node B (Node A Leaf Leaf) (Node C Leaf Leaf))
True
*Main> A <= C
False
```

What is the problem here? The problem is not (really) with the code `isSorted`, but rather with the `Ord` instance for `ABC`: It does not behave as we would expect it to, in particular, it is not transitive:

```
*Main> (A <= B && B <= C, A <= C)
(True,False)
```

If we look at the documentation of the `Ord`[25] type class, we see that any instance of `Ord` is expected to be, among other things, transitive. Indeed, most type classes come with additional requirements, or *laws*, that should hold for its instances.

In this sense, the constraint in the type signature of `isSorted` should not be read as "for any type `a` that implements the signature of the `Ord` type class...", but rather as "for any type `a` that is ordered...". Not the *interface* matters, but rather the *semantic meaning* behind it.

Conversely, if you come across a type class without any semantic meaning, i.e. one that just overloads a name, then that is clearly fishy. Nothing good will come out of a type class like, say

```
class IntAble a where toInteger :: a -> Integer
```

if it does not also come with an abstract meaning that should be shared by all instances.

By the way, did you notice the `Eq` constraint in the head of the declaration of the `Ord` type class:

```
class Eq a => Ord a where
```

This means that only types that are an instance of `Eq` may have an instance of `Ord`. The upshot is that a function with a `Ord a` constraint may also use `(==)`, without explicitly listing the `Eq` constraint.

Exercise 22

Look up the `Semigroup` type class, and find its laws.

Exercise 23

Can you think of a `Semigroup (Tree a)` instance? Or maybe even more than one? How can you be sure it is a lawful instance?

Exercise 24

[25] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Ord

The `Monoid` class[26] extends the `Semigroup` class with an operation `mempty :: Monoid a => a` that is supposed to be a neutral element of `(<>)`.

Given a function signature `summarize :: Monoid a => Tree a -> a`, can you guess what it does? What would be the implementation you expect?

With that implementation, can you use `summarize` to distinguish trees that differ in shape, but have the same elements in the same order? What does this imply for search trees?

5.4 Type-driven code synthesis

Let us turn to the most sophisticated use of type classes (within this lecture): Type driven code synthesis! Again, this is a fancy word for something rather simple, but it is very powerful, and a driving idiom for many problems in the Haskell space.

In the previous chapter we have seen a number of common types that can be combined to build larger types – `Maybe`, tuples, `Either`, list, etc. If there is functionality that we want to provide at many different types, then we can use type classes to describe how to get that functionality for each of these building blocks, assuming we have it for their argument types, and then the user automatically gets the functionality for any complex type they build.

Typical examples for such functionality are parsers, pretty-printers, serialization libraries, substitutions of sorts, random values, test case generation...

Let us define a type class that describes *finite* types, i.e. types with a finite number of values, and a function that returns the number of values in the type:

```
class Finite a where
    size :: Integer
```

Is this a good type class, i.e. does it have *meaning*? Yes, it does: When a type has an instance of `Finite`, it means that the type is finite, and that `size` designates the number of elements. Unfortunately Haskell does not prevent you from writing an instance that does not adhere to that, but that would then simply be wrong.

This definition is a bit weird: The type signature of `size` does not mention the type `a` anywhere. On the one hand, this makes sense: We do not need a concrete element in our hand to ask the question "how many elements are there in `a`". On the other hand, when we use `size` somewhere, how will the compiler know for which type we want to invoke it?

Therefore, this would be prohibited in plain Haskell. But as mentioned before, contemporary Haskell often uses language extensions supported by the compiler, and that is what we will do here, namely

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE ScopedTypeVariables #-}
```

(in `ghci` you can type `:set -XAllowAmbiguousTypes -XTypeApplications -XScopedTypeVariables`). With these extensions, the class declaration is accepted, and we can use the syntax `size @Bool` to say which instance to use.

[26] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Monoid

We start with instances for some basic types:

```
instance Finite Bool where size = 2
instance Finite () where size = 1
data Suit = Diamonds | Clubs | Hearts | Spades
instance Finite Suit where size = 4
```

More interesting are the instances for the types that build on other types. These are not always `Finite`, but only if the types therein are themselves `Finite`, so we constrain the instance itself:

```
instance Finite a => Finite (Maybe a) where
    size = size @a + 1
instance (Finite a, Finite b) => Finite (a,b) where
    size = size @a * size @b
instance (Finite a, Finite b) => Finite (Either a b) where
    size = size @a + size @b
```

Even the function type is finite, if both domain and codomain are finite:

```
instance (Finite a, Finite b) => Finite (a -> b) where
    size = size @b ^ size @a
```

Note that for obvious reasons we do not have an instance for `Integer`, or the list type.

Now that we have sown the seed, we want to reap the fruit: Whatever complex type we build out of these constructors, we can evaluate `size` that that type:

```
Prelude> size @(Maybe Bool)
3
Prelude> size @(Maybe (Maybe Bool))
4
Prelude> size @(Suit -> Bool)
16
Prelude> size @(Suit -> Bool, Bool -> Suit)
256
Prelude> size @((Suit -> Suit) -> Maybe Bool)
139008452377144732764939786789661303114218850 8
085291379916048244300360726297664359410017691 5
410960952181166554054889943552 1
```

The utility of such a `size` function is questionable (but not completely void), but I hope you understand the power behind this approach, and also recognize the pattern if you see in the wild.

In fact, instances of the `Eq` and `Ord` class for the container types like tuples and lists etc. also follow this pattern.

5.5 Common pre-defined type classes ⋆

You should know the following common type classes. Follow the links for the list of methods and other documentation:

- `Eq`[27]: Equality (or equivalence)
- `Ord`[28]: Ordering
- `Num`[29]: Numeric operations (`(+)`, `(-)`, `(*)` and others). There are more numerical type classes (`Real`, `Integral`, `Fractional`, `RealFloat`).
- `Show`[30]: Provides `show :: Show a => a -> String` to serialize a value to a textual representation that is (supposed to be) valid source code. Should be used for debugging mostly, or to convert numbers to strings.
- `Read`[31]: Provides `read :: Read a => String -> a`, which goes the other way. Again, not ideal for production use, but can sometimes be used with `Show` to scaffold serialization. If you have to use it, consider using `readMaybe`[32].
- `Functor`[33]: Provides `fmap :: Functor f => (a -> b) -> f a -> f b`. This type class can only be instantiated for type constructors (`Maybe`, the list type, etc.). It provides the ability to apply a function to each value within the container (for types that are a container, of sorts).
- `Applicative`[34] and `Monad`[35] are used to model effects of sorts, for example to hide bookkeeping (in a parser) or safely allow side-effects (in IO code). We will discuss them in great detail in the next chapter.
- `Foldable`[36] and `Traversable`[37] are abstractions over containers where elements can be visited in sequence, i.e. a generalization of lists.

6 Monads

Haskell is famous for *monads*. If you are scared by complicated sounding words, you might even consider them to be infamous. Judging by the number of "monad tutorials" and other noise about this topic, these monads must indeed be crazy arcane black magic.

So what's the deal? What is a monad?

Really, the concept of a monad is surprisingly small. It is a pattern of abstraction, expressed as a type class with merely two essential methods and a small number of laws. That's it, the rest is just applications. But this small idea turns to be amazingly powerful and expressive.

I can't help but notice that monads are like burritos: What is a burrito? It is a bunch of protein and seasoning, neatly wrapped in a flour tortilla. That's it. But with just that knowledge, it is impossible to fully appreciate or recreate the wealth and richness of Mexican cuisine. The idea is simple, but the applications are rich and manifold and, and therefore require skill and experience to master.

[27] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Eq
[28] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Ord
[29] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Num
[30] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Show
[31] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Read
[32] http://hackage.haskell.org/package/base/docs/Text-Read.html#v:readMaybe
[33] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Functor
[34] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Applicative
[35] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Monad
[36] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Foldable
[37] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Traversable

(Oh, and of course, sometimes a taco would do better than of a burrito. Monads are not always the right tool.)

This is a cute analogy, but it does not help the aspiring Haskell reader. So how to we proceed from here? This material offers two choices:

- A quick path to understanding "imperative Haskell code", i.e. Haskell code that uses the `IO` monad and `do` notation, and looks similar to, say, Python code. This path avoids almost all technical details about monads, and simply gives you a way to decipher the syntax. This is in the chapter on imperative Haskell.
- A slow path where we actually look at the `Monad` type class, the idea behind it, and some of the more advanced (but still common) applications of it. This is this chapter.

6.1 Kinds

Before we talk about monads, we have to discuss the concept of *kinds*.

You might have already noticed that there is a fundamental difference between a type like `Bool` and a type like `Maybe`. The former has values, e.g. `True`, and it can be an argument or a return value of a function. That is not true for `Maybe`. There is no value that has type `Maybe`! Only when we say what type we maybe have, it gives us a proper type. So `Maybe` is not a normal type in that sense, but `Maybe Bool` is, or in general `Maybe a` for any normal type `a`.

So what is `Maybe`? It is a *type constructor*! It takes a normal type (with values, like `Bool`) and constructs a new normal type from it (namely `Maybe Bool`).

We can apply `Maybe` multiple times: `Maybe (Maybe Bool)` is also a normal type. But we cannot apply `Maybe` to itself: `Maybe Maybe` is nonsense.

This is all very similar to the term level, where `True` is a Boolean, but `not` is not a Boolean. But `not` can be applied to a Boolean, and `not True` is another Boolean. We can apply it multiple times `not (not True)`, but we cannot apply it to itself, `not not` is nonsense.

On the term level, terms have *types* that describe what compositions make sense and which compositions are disallowed. `True` has type `Bool`, and `not` has type `Bool -> Bool`, which explains why you can apply `not` to `True`, but not to `not`.

We find the same on the type level: Types have *kinds* that describe which compositions make sense and which compositions are disallowed. `Bool` has kind `*` (pronounced "star" or simply "type"), and `Maybe` has kind `* -> *`, which explains why you can apply `Maybe` to `Bool`, but not to `Maybe`.

The kind `*` is the kind of all the normal types, which have values, and which can be the argument or return type of a function. `* -> *` is the kind of simple *type constructors* like `Maybe`, or `Tree`, or the list type. Some have more than one argument, e.g. `Either` has kind `* -> * -> *`. GHCi happily tells you the kind of a type constructor using the `:kind` command:

```
Prelude> :kind Bool
Bool :: *
Prelude> :kind Maybe
Maybe :: * -> *
Prelude> :kind Either
```

```
Either :: * -> * -> *
```

In mundane code, kinds do not get more complicated than that, but there are good uses for higher kinds, such as

```
Prelude> newtype Fix f = Fix (f (Fix f))
Prelude> :kind Fix
Fix :: (* -> *) -> *
```

6.2 The kind of type classes

Type classes are also kinded. You can have a constraint `Eq Bool`, or `Eq (Maybe Bool)`, but not `Eq Maybe` – that does not make sense. So `Eq` takes a parameter of kind `*`, and produces a constraint:

```
Prelude> :kind Eq
Eq :: * -> Constraint
Prelude> :kind Monoid
Monoid :: * -> Constraint
```

Most type classes that we saw in the previous section have that kind (`Eq`, `Ord`, `Num`, `Show`, `Read`, `Monoid`). But there are also type classes that characterize type constructors, in particular the infamous `Monad`:

```
Prelude> :kind Functor
Functor :: (* -> *) -> Constraint
Prelude> :kind Monad
Monad :: (* -> *) -> Constraint
```

6.3 A close look at `* -> *`

So according to the kind of `Monad`, only type constructors of kind `* -> *` can be instances of `Monad` . And I believe that one way towards grasping monads is to get a good handle on the concept of a type constructor, both concretely, and the abstract concept. Once we have seen enough concrete examples of them, worked out the common patterns, we can appreciate that *monad is an abstraction over type constructors*.

Let us start abstractly, and consider a type constructor `m` with kind `* -> *`. So for every type `a` (of kind `*`), there is a type `m a`. The meaning of `m a` is (usually) very different from, but still somehow related, to the meaning of `a`. It could be "extra data", it could be additional behavior, some kind of bookkeeping, or effects of sorts.

To make this more concrete, let us look at some examples of type constructors with kind `(* -> *)`, and how they change the meaning of a type.

```
data Maybe a = Nothing | Just a
data Either e a = Left e | Right a
data [a] = [] | a : [a]
newtype Identity a = Identity a
data Proxy a = Proxy
newtype Reader r a = Reader (r -> a))
newtype State s a = State (s -> (a, s))
```

```
newtype Parser a = Parser (String -> [(a,String)])
data IO a =  ¯\_(ツ)_/¯
```

- The first example is the `Maybe` type that we have looked at before. A value of type `Maybe a` is either `Just` a value of type `a`, or it is `Nothing`. So `Maybe` takes a type `a` and adjoins an extra element to it.

  You can think of `Maybe a` as a container with zero or one element. But you can also think of it as a *computation* that can fail, or return an `a`. This is commonly used for lookups in maps, or text parsing that can fail.

  `Maybe` has kind `* -> *`, so it "fits" the `Monad` type class, and an `instance Monad Maybe` would make sense.

- The next example, `Either`, is a close relative of `Maybe`. It also commonly models "a with failure", but the failure can carry additional data of type `e`.

  Because it takes two type arguments, the kind of `Either` is `* -> * -> *`, and there cannot be an `instance Monad Either`. But we can make it fit by providing *one* type argument. For each type `e`, `Either e` has kind `* -> *`, and it would make sense to have an `instance Monad (Either e)`.

- The list type has kind `* -> *`. The meaning of `[a]` is "zero, one or more values of type `a`" and shows that it is *not* the case that a value of type `m a` always contains a value of type `a`.

  This example becomes immediately more interesting if we *interpret* a list differently. Instead of considering it to be a container of many values, we can also think of it as a non-deterministic computation, i.e. a computation with multiple possible results of type `a`.

- A bit vacuous, but we can create a type constructor that does not actually change the meaning of the type; that would be the `Identity` type constructor.

- Since we are looking at corner cases, let us introduce the `Proxy` type constructor. It is an example where `m a` really never contains a value of type `a`. In fact, a value of type `m a` is always just `Proxy`. The type `a` is merely a phantom that spooks on the type level.

- Back to more useful examples: The meaning of `Reader r a` is "a value of type `a` – if you give me an `r`". Again, no value of type `a` is contained in `Reader r a`; if anything, there is a promise for one. Or, actually, for many: it has one for each possible value of type `r`.

  Incidentally, the `Reader` type is just an alternative name for the function arrow `(->)`, and if we partially apply the function arrow to a type, i.e. `(->) r`, we get something of kind `* -> *` just as well.

- A similar idea is behind the `State` type constructor. Here, a value of type `State s a` is "a value of type `a`, if you give me an `s`, and by the way, I will also give you a new value of type `s`", or, maybe more helpfully, "a computation that accesses state of type `s` and produces a value of type `a`".

- The `Parser` type constructor implements a backtracking parser. Let's not looking at the definition too much, and just appreciate that going from `a` to "something that can parse a value of type `a`" can be modeled as a type constructor.

- And finally there is the `IO a` type constructor, for which the definition is opaque. But it still has a clear meaning: A value of type `IO a` is a computation that, after interacting with the external world (terminal, files, network, randomness, etc.), produces a value of type `a`. (See the chapter on imperative code if you want a diversion.)

This was a long list, and I could have easily extended it with many more. So what is the point? The point is that there are a large number of very different concepts that can be naturally expressed as a type constructor. If we now venture out to find similarities between them, we will stumble upon monads.

6.4 The `Monad` type class

If we look at the list above, we might notice that all of them have something in common: When working with these objects, we often want to *compose* in the following way.

- If I have a computation that returns an `a` or fails, and one that takes an `a` and may fail or return a `b`, I want to plug them together to get a `b` (or failure).
- If I have a computation that accesses state and produces a value of type `a`, and one that – given a value of type `a` – accesses state and produces a value of type `b`, I want to compose them to get a computation that accesses state and returns a `b`.
- If I have a parser that parses a number, and a parser that parses a string of a given length, I want to compose them to parse a string-with-length data format.
- If I have a computation that reads from a file and returns the content as a string, and one that takes a string and writes it to another file, I want to compose them to one that copies a file.
- etc.

We could implement this composition separately for each of these, and obtain a long list of functions

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThenEither :: Either e a -> (a -> Either e b) -> Either e b
flipConcatMap :: [a] -> (a -> [b]) -> [b]
bindIdentity :: Identity a -> (a -> Identity b) -> Identity b
bindProxy :: Proxy a -> (a -> Proxy b) -> Proxy b
bindReader :: Reader r a -> (a -> Reader r b) -> Reader r b
bindState :: State s a -> (a -> State s b) -> State s b
bindParser :: Parser a -> (a -> Parser b) -> Parser b
bindIO :: IO a -> (a -> IO b) -> IO b
```

This would suffice for many applications, but it is not nice – I even ran out of names to use.

Haskell is all about abstraction, and clearly, there *is* a common pattern here. If distill the abstract type here, using `m` to stand for the type constructor, we get `m a -> (a -> m b) -> m b`. Remember that for a moment.

For all these type constructors, there is something else that we want to do. When we have a value `x` of type `a`, we want to be able to treat it as

- a computation that could fail (but doesn't), or
- a computation that could access state of type `s` (but doesn't actually look at it) before returning `x`,
- a parser that does not touch the input, but simply always produces `x`,
- etc.

Again, we could have separate functions for each of these injections, but that would be tedious. So if we try to phrase this as an abstract type, in terms of `m`, we get `a -> m a`.

This brings us to the concept of a monad: A type constructor that allows composition and injection in this way is a monad.

6.4.1 The definition

Finally, this is time to look at the actual definition of the `Monad` type class[38] (with optional and obsolete methods omitted):

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

-- Laws:
-- return a >>= k = k a
-- m >>= return = m
-- m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

The most important bit of information is actually only implicit: The fact that the argument `m` has kind `* -> *`. We can infer that from the fact that `m` is applied to `a` in the type signature of the methods.

6.4.2 The operations

What we do find very explicitly, however, are the two type signatures that we have just distilled:

- The operator with the funny arrow does the kind of composition that we wanted. It is also called *bind* or the *monad composition operator*.
- The function `return` (also available as `pure`) injects a value into the monad.

In this context, when `m` is a monad, we call a value of type `m a` "a monadic action returning a value of type `a`", or sometimes also "monadic computation with return type `a`".

For additional intuition, squint at these type signatures, and imagine the `m` were not there:

- Then the bind operator would simply have type `a -> (a -> b) -> b`, and due to parametricity it is clear what such an operator does: It is function application! (i.e. `($)` with parameters flipped, also available as `(&)`[39] in `Data.Function`.) The bind operator does morally the same, while at the same taking care of the particular meaning that is introduced by the type constructor `m`.
- Similarly, `return` has squinted type `a -> a` and thus simply becomes the identity function. So `return` doesn't do anything interesting to its argument, it just makes it look like it *could* have additional meaning.

6.4.3 The laws

As discussed before, a type class is more than just a set of function signatures: The laws that come with it are equally important.

- The first two laws can be summarized as "`return` is the neutral element of bind". Notice how `return` is a function that perfectly fits as the second argument to bind? If we do so, the bind operator composes the particular meaning of the first argument of type `m a` with the particular meaning introduced by `return`. The second law says that this this does *not* actually affect the meaning, we

[38] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Prelude.html#t:Monad
[39] http://hackage.haskell.org/package/base/docs/Data-Function.html#v:-38-

could have just used the original `m a`. In that sense, `return` turns an `a` into something that has the shape of an `m a`, without giving it any interesting meaning beyond the value of type `a` itself.

- The last law expresses that the bind operator is associative. Composing monadic operations – `m`, `k` and `h` in the law – we get the same result independent of whether we combine `m` with `k` first or `k` with `h` first.

  Note that the *order* of these operations still matters – in general, the bind operator is not commutative.

6.4.4 No escape?

Let me point out two things that are notably missing from this type class:

1. There is no method that takes a monadic action `m a` and returns just the `a`. Hence the slogan "You can't escape the monad.".

   The slogan is only half true, and should better be "You can't escape an arbitrary monad", as you can escape concrete monads (think of `Identity`, or `Reader r`, if you supply the `r`). But in general it is not possible; think of `Proxy` or, most confusingly for novice Haskell writers, `IO`.

2. There is no method that creates any *interesting* monad actions. We already deduced that the `return` method cannot actually do anything interesting, and the bind operator merely combines what is there.

This shows that the monad abstraction is *only* concerned with the composition of the monadic actions, but to do anything interesting, we need additional functions for a concrete monad.

6.5 More monad operations ⋆

It suffices to define `return` and `(>>=)` to define a monad, but when working with monads, there a number of other operations in common use:

```
(>>)    :: Monad m       => m a        -> m b -> m b
fmap    :: Functor f     =>   (a -> b) -> f a -> f b
(<$>)   :: Functor f     =>   (a -> b) -> f a -> f b
(<$)    :: Functor f     =>   a        -> f b -> f a
(<*>)   :: Applicative f => f (a -> b) -> f a -> f b
(<*)    :: Applicative f => f a        -> f b -> f a
(*>)    :: Applicative f => f a        -> f b -> f b
liftA2  :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
(>=>)   :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
join    :: Monad m => m (m a) -> m a
mapM    :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_   :: Monad m => (a -> m b) -> [a] -> m ()
forM    :: Monad m => [a] -> (a -> m b) -> m [b]
forM_   :: Monad m => [a] -> (a -> m b) -> m ()
when    :: Applicative f => Bool -> f () -> f ()
unless  :: Applicative f => Bool -> f () -> f ()
forever :: Applicative f => f a -> f b
```

For now, pretend that instead of `Applicative` or `Functor` it would read `Monad`, we will discuss the difference later.

- The first bunch of operators are simply for various combinations of
  - is an argument an action or a pure value
  - is one argument a function? If not, which value is used.

  and you tell which it is from the type. Note that `<$>` is simply a different name for `fmap`.

  Here are concrete examples:

  ```
  getName :: IO String
  getName = putStr "What is your name? " >> getLine

  -- parses the format `23+42i` as a complex number
  parseComplex :: Parser Complex
  parseComplex = C <$> parseInteger <* parseStr "+" <*> parseInteger <* parseStr "i"
  ```

  Exercise 25

  Give definitions of the operators up to `join` using `(>>=)`, `return`, and operators you already defined. (You will have to change the constraints to `Monad` for this to typecheck.)

- The `(>=>)` operator is very similar to `(>>=)`, only that both arguments have the same type. With this operator, the monad laws are much more accessible:

  ```
  return >=> m = m
  m >=> return = m
  (a >=> b) >=> c = a >=> (b >=> c)
  ```

- `mapM` and `forM` apply a monadic action to each element of a list, collecting the results. The variants with underscore do not collect the results, e.g. when you are only interested in the monadic effect, and are more efficient.

  These functions demonstrate one benefit we get from monads that goes beyond some petty operator overloading: It distills the (still simple, but non-trivial) concept of "traversing a list while doing *something* on the side". These functions are defined once, and we can use it in all those different contexts, whether that is handling failures, manipulating state, consuming parser input and backtracking or exploring all possibilities of non-deterministic computation.

- `when` and `unless` are pretty simple one: If the Boolean argument is true (respectively false), the action is executed, otherwise not.

  Exercise 26

  Why is the type not `when :: Bool -> m a -> m a`?

- `forever` just keeps executing the same action over and over. This does not make sense for every monad, but it is useful for some – including `IO`, where you might find an event loop wrapped in `forever`. The return type `b` is completely unconstrained, because `forever` never "returns" anyways.

6.6 do notation ⋆

So monads are powerful and ubiquitous, and we have these expressive monad operators to compose monadic actions, both for a concrete monad, or abstractly. Code written using these operations might look like this:

```
-- parses the format `23+42i` as a complex number
parseComplex :: Parser Complex
parseComplex =
  parseInteger >>= \r ->
  parseStr "+" >>= \_ ->
  parseInteger >>= \i ->
  parseStr "i" >>= \_ ->
  return (C r i)

-- parses a sequence of such numbers, with the length given before
parseSequence :: Parser [Complex]
parseSequence =
  parseInteger >>= \n ->
  forM [1..n] (\_ -> parseComplex)
```

This works, but it is not really pretty. Therefore, Haskell offers syntactic sugar for working with monads that hides the (>>=) operator. It is called do-*notation* and makes the code read almost like imperative code:

```
-- parses the format `23+42i` as a complex number
parseComplex :: Parser Complex
parseComplex = do
  r <- parseInteger
  parseStr "+"
  i <- parseInteger
  parseStr "i"
  return (C r i)

-- parses a sequence of such numbers, with the length given before
parseSequence :: Parser [Complex]
parseSequence = do
  n <- parseInteger
  forM [1..n] (\_ -> parseComplex)
```

But remember, it is really just sugar, and code involving do simply gets translated into code using >>=. The translation is pretty straight-forward, and essentially as follows:

```
do x <- a        a >>= (\x -> do more)          do a             a >>= (\_ -> do more)
   more                                             more

do let x = e     let x = e in do more           do a             a
   more
```

Note the difference between `let x = e` and `x <- a`. The former is simply a pure `let`, i.e. gives a name to a pure expression; no monadic actions are executed here, no bind is involved. The latter invokes (>>=)

and x is bound to the "result" of that monadic actions.

When use use this style with the `IO` monad we end up with code that almost looks like normal, say, Python:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile from to = do
    content <- readFile from
    putStrLn ("Read " ++ show (length content) ++ " bytes.")
    writeFile to content

main :: IO ()
main = do
    putStrLn "Which file do you want to copy?"
    from <- getLine
    putStrLn "Where do you want to copy it to?"
    to <- getLine
    copyFile from to
    putStrLn "Done copying."
```

Exercise 27

Implement `forM :: Monad m => [a] -> (a -> m b) -> m [b]` using do-notation.

6.7 `Functor` and `Applicative` ⋆

In the listing of derived monad operations, some type signatures had a `Functor` or `Applicative` constraint instead of `Monad` constraint. What are these?

Well, they are type classes, and here are their definitions, including laws

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- Laws:
-- fmap id = i
-- fmap f . fmap g = fmap (f . g)

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

-- Laws:
-- pure id <*> v = v
-- pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
-- pure f <*> pure x = pure (f x)
-- u <*> pure y = pure ($ y) <*> u
```

Like `Monad`, these are type classes of kind `(* -> *) -> Constraint`, i.e. `f` is a type constructor.

- The `Functor` type class allows you to apply a (pure) function to the (well, all) `a` in an `f a`. This is most intuitive when `f` is a container data structure like `Maybe` or lists. In fact, if `f` is simply the list type constructor, then `fmap` is the normal `map` function. For a type constructor with a more computational interpretation, such as `Parser` or `IO`, `fmap` simply "keeps the extra meaning, and applies the function to the result".

  The laws also express that `fmap` does not actually change the extra meaning of its argument.

- The `Applicative` type class is very similar to `Monad`: It has `pure` instead of `return`, but they are essentially the same. And it has a binary operation, sometimes called *ap*, that takes two values with extra meaning and composes them. The crucial difference is: With `(>>=)`, you choose which action to do as the second argument based on the *return value* of the action of the first argument. With `(<*>)`, both actions need to be given separately and independently, and only their results are combined. Therefore, there is strictly less you can do with the `Applicative` interface than with the `Monad` interface.

  The laws are analogous to the monad laws: Essentially, `(<*>)` is associative and `pure` is its unit.

Every `Monad` is both an `Applicative` and a `Functor`, as we have seen in the exercise in the previous section. So `Monad` is more expressive than the other two. Why do we bother with separate type classes for them? Because less expressive interfaces are more widely applicable! If the user of an interface can do less, then the implementor of an interface has more freedom.

What can we do with that freedom? Create more instances!

- Consider the following type constructor:

  ```
  data Two a = Two a a
  ```

  It is a container, not unlike `Maybe` and lists, that stores exactly two values of type `a`. It is impossible to have a (lawful) `Monad` instance for this type. But it can have a `Functor` instance, and it is a very useful instance that we would often want:

  ```
  instance Functor Two where fmap f (Two x y) = Two (f x) (f y)
  ```

  This type constructor can also be given an `Applicative` instance, so it serves as a proof that `Applicative` is more general than `Monad`.

- To see that `Functor` is more general than `Applicative` we can use the following type:

  ```
  data Tagged t a = Tagged t a
  ```

  that is a value `a` with a tag `t`. This is easily a functor (just apply the function to the second parameter of `Tagged`), but there is no function `pure :: a -> Tagged t a`, because it would have to come up with a value of type `t` out of thin air, so there cannot be an `instance Applicative (Tagged t)`.

  Exercise 28

  You might observe that `Tagged` is simply the existing pair type. There is an `Applicative` instance for pairs. Look it up! How does that fit to what I just said?

- And since we are discussing counter-examples, the type constructor

  ```
  newtype Printer a = Printer (a -> String)
  ```

  is not even a `Functor`: Since a `Printer a` *expects* a value of type `a`, you can't apply a function `a -> b` anywhere. (You could apply a function `b -> a`; this means that `Printer` is a *contravariant functor*.)

At a high level, the main advantage of an applicative computation over a monadic is that the *structure of the computation* is static, and can be known ahead of times. This is not the case in a monadic computation, because the second argument to `(>>=)` is an opaque function, and only once a concrete a was extracted from the first argument do we know what kind of computation is produced by the second.

As a concrete example, consider a library that provides a type constructor `AParser` that is an instance of `Applicative`, but *not* of `Monad`. You write a parser `p :: AParser a` using this interface. Then the library can "run" this parser without actual input, and learn everything it *would* do. It could use that to provide a function `describe :: AParser a -> String` that produces a grammar for the given parser, which you can use as documentation, and thus these can never go out of sync.

Solutions

Solution 1

Both operators are left-associative, so that `10 - 2 + 3 - 4` means `((10 - 2) + 3) - 4` as expected.

Solution 2

The precedences of `(+)` and `(-)` are the same, and smaller than the precedence of `(*)`, which is again smaller than the precedence of `(^)`.

Solution 3

`7`

Solution 4

`3`

Solution 6

```
absoluteValue x = if x < 0 then - x else x
```

Solution 7

```
isHalfRound x = x `mod` 10 == 0 || x `mod` 10 == 5
```

Solution 8

```
isEven x = x `mod` 10 == 0 || x `mod` 10 == 2 || x `mod` 10 == 4 || x `mod` 10 == 6 || x `mod` 10 == 8
```

Solution 9

```
sumDigits n = if n < 10 then n else sumDigits (n `div` 10) + (n `mod` 10)
```

Solution 10

```
fixEq f x = if x == f x then x else fixEq f (f x)
```

Solution 11

```
isMultipleOf3 x = fixEq sumDigits x == 3 || fixEq sumDigits x == 6 || fixEq sumDigits x == 9
```

Solution 12

The definitions for `countCountDigits` and `sumSumDigits`:

```
Prelude> countCountDigits = twice countCountDigits
Prelude> sumSumDigits = twice sumDigits
```

Solution 13

```
id :: a -> a
```

Solution 14

It is the function `flip` that takes a function and swaps its first two arguments.

Solution 15

This type is legal. Every value of type `Wat` is built from the constructor `Wat`, applied to another value of type `Wat`. So, unless there are exceptions or nontermination around, it is just an infinite tower of `Wat`s:

```
wat :: Wat
wat = Wat wat
```

Solution 16

There are four: `Nothing`, `Just Nothing`, `Just (Just False)` and `Just (Just True)`. It can be useful if, for example, the outer `Maybe` indicates whether some input was *valid*, whereas `Just Nothing` could indicate that the input was valid, but empty. But arguably this is not best practice, and dedicated data types with more speaking names could be preferred here.

Solution 17

```
Maybe (Integer, Integer)
```

Solution 18

```
fromEitherUnit :: Either () a -> Maybe a
fromEitherUnit (Left ()) = Nothing
fromEitherUnit (Right x) = Just x

toEitherUnit :: Maybe a -> Either () a
toEitherUnit Nothing = Left ()
toEitherUnit (Just x) = Right x
```

We can make `fromEitherUnit` more polymorphic; it can simply ignore the argument to `Left`. We cannot do this in `toEitherUnit`: we would not have a value of type `b` at hand to pass to `Left`.

Solution 19

```
Pondering the question...
42
```

The line `return 23` doesn't do anything: There is no side-effect, and the result (the value `23`) is not bound to any variable and hence ignored.

Solution 20

```
Hooray!
Hooray!
Hooray!
Almost done
```

```
Hooray!
Done
Done
```

Note that the `putStrLn "And up she rises."` is never executed.

Solution 21

```
instance Eq Employee where
    e1 == e2 =
        name e1 == name e2 &&
        room e1 == room e2 &&
        pubkey e1 == pubkey e2
```

The problem is: If the record gains additional fields, this code still compiles, and the programmer is not warned that they should update it. By not using the record accessors, and using normal constructor syntax instead, this can be avoided:

```
instance Eq Employee where
    Employee n1 r1 p1 == Employee n2 r2 p2 =
        n1 == n2 && r1 == r2 && p1 == p2
```

Solution 22

The `Semigroup` type class is defined as

```
class Semigroup a where
  (<>) :: a -> a -> a
```

with the additional requirement that the `(<>)` operation is associative.

Solution 23

There are (at least) two sensible instances for the `Semigroup` type class for trees:

1. The first one concatenates two trees, so that an in-order traversal first visits the value of the left and then of the right tree:

   ```
   instance Semigroup (Tree a) where
       Leaf <> t = t
       (Node x l r) t = Node x l (r <> t)
   ```

   There are variations of this code that are more likely to produce a balance tree – although then it might be that associativity holds when one considers different shapes of the same data equivalent (which is commonly the case for search trees).

2. Another one traverses both trees in parallel, using a `Semigroup` instance for the elements to combine values that are present in boht trees:

   ```
   instance Semigroup a => Semigroup (Tree a) where
       Leaf <> t = t
       t <> Leaf = t
       Node x l1 r1 <> Node y l2 r2 = Node (x <> y) (l1 <> l2) (r1 <> r2)
   ```

   Here, the instance head itself has a constraint: This instance for `Tree a` only exists if there is a `Semigroup` instance for the type of values.

Which instance is the right one? That depends on the purpose of the `Tree` data structure in the code; and maybe neither is the right one, in which case it might be better to have *no* instance at all, and use normal functions for these operations.

Solution 24

One might expect this code:

```
summarize :: Monoid a => Tree a -> a
summarize Leaf = mempty
summarize (Node x l r) = summarize l <> x <> summarize r
```

(at least given everything we did so far considered in-order traversals.)

If we have two trees with the same elements in the same order, but in a different shape, e.g.

```
t1 = Tree x Leaf (Tree y Leaf (Tree z Leaf))
t2 = Tree y (Tree x Leaf Leaf) (Tree z Leaf Leaf)
```

for some values `x`, `y` and `z`, then we will always have `summarize t1 = summarize t2`, assuming that the `Monoid` instance for the type of values is lawful. This is important if we use these trees as search trees, where the internal shape should be an implementation detail that should not be visible from the outside.

Solution 27

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM [] f = return []
forM (x:xs) f = do
    y <- f x
    ys <- forM xs f
    return (y : ys)
```

Solution 25

```
import Prelude hiding (fmap, (<$>))
(>>) :: Monad m => m a -> m b -> m b
a >> b = a >>= (\_ -> b)
fmap :: Monad f => (a -> b) -> f a -> f b
fmap f a = a >>= (return . f)
(<$>) :: Monad f => (a -> b) -> f a -> f b
(<$>) = fmap
(<$) :: Monad f => a -> f b -> f a
x <$ a = const x <$> a
liftA2 :: Monad f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = a >>= (\x -> b >>= (\y -> return (f x y)))
(<*>) :: Monad f => f (a -> b) -> f a -> f b
(<*>) = liftA2 (\f x -> f x)
(<*) :: Monad f => f a -> f b -> f a
(<*) = liftA2 (\x y -> x)
(*>) :: Monad f => f a -> f b -> f b
(*>) = liftA2 (\x y -> y)
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
a >=> b = (\x -> a x >>= b)
```

```haskell
join :: Monad m => m (m a) -> m a
join a = a >>= id
```

Solution 26

Since `when False action` is not supposed to execute `action`, it has no way of producing a `m a`. But it can always create a `m ()` using `return ()`.

Solution 28

The instance is

```haskell
instance Monoid r => Applicative ((,) r)
```

and the `Monoid` constraint on `r` gives us exactly the operations needed to to implement `pure` and `(<*>)`. Try it yourself!