# Speeding Through Haskell

**With Example Code!**

Mihai Radu Popescu

questions@sthaskell.com

To `#haskell`, where all questions are answered in majestic stereo.

# Contents

# Part I.

# Starting Out

# 1. Introduction

The Haskell community has an acute shortage of buggy underdocumented programs.

*(sorear)*

## 1.1. About the Book

### 1.1.1. This is a work in progress.

***Warning!*** This book is a work in progress. Read at your own risk!

Hello there! This is a book that will show you around the Haskell programming language. If you're not already familiar (or too familiar) with programming in another language, you might need to put in extra work. Don't be discouraged! While the stuff in the beginning may seem extremely boring, mind-blowing things start happening later on.

This book has a lot of footnotes. You don't have to read them, but sometimes you might gain some insight by doing so. You can click on them (do it here[1]) to jump to them faster (readers from the website might want to download the book for this reason). You can click on the table of contents as well.

Feel free to jump around the book as well! I've added as many links as possible to help you get around to cooler stuff if you're curious about it. I also have short recaps of the important stuff as it's needed so you don't have to go back and hunt for the particular chapter the concept got introduced in.

The writing in this book may not be polished yet, and some things may be missing, but take a look — you might just like it!

### Your turn! Exercises

At the end of each subsection I will add some exercises in the form of questions, quizzez or whatever. If you're stuck, as the saying goes, Try Harder$^{\text{TM}}$. I usually go to Google when I'm not sure of something. Don't be afraid to "cheat" this way[2] but be sure that you learn something from it!

After you're done with the exercises, read the hints at the end. They often provide additional insight beyond answering the question. Some might explain why we do things one way and not another. Others might give interesting or fun facts about Haskell. Or you might just have found a solution that is completely original and creative!

## 1.2. Why Haskell?

### 1.2.1. Who might want to learn

Every language (human or computer) is unique. But there exists a special breed of languages — those that challenge and shape the way one thinks. Haskell is one of them — lost innovation in a sea of clichés. Unfortunately, the only people apparently interested in Haskell are academics who blindly push the boundaries and gurus who want to learn "just one more language".

[1] If it didn't work, you might want to download the book (google docs link). If it still doesn't work, get Adobe Reader.

[2] It's not cheating! Googling stuff is an amazing way to get unstuck, read more about things and learn new and better ways of solving problems. I do it all the time.

On a more concrete note, if Haskell were to have a list of prerequisites, it would be very unusual indeed — at least two of the following:

- Extensive programming experience
- A background in mathematics
- An inclination towards the abstract
- Perseverence
- Hard work

### 1.2.2. For Programmers

I never intended to (and still don't quite) take programming seriously. I wanted something quick, fun and challenging to kill some time, clear my thoughts and, above all, stop performing repetitive tasks. My first language was Python — easy, fun, good with the teachers. After about two weeks, I let it go and tried others: Common Lisp, C, Perl, Java, and finally, I fell in love with Haskell.

One might say Haskell is a bit different. For example, in Haskell:

- `return` doesn't return
- Classes aren't really classes
- "Variables" are actually constants.
- The code *might not* execute in the order shown on the screen.

Below are some of my favorite snippets of code, each on a separate line. They're classics, and really show how Haskell stands out.

```haskell
fibonacci = 0:1:zipWith (+) fibonacci (tail fibonacci)
primes = nubBy (\x y -> (gcd x y) > 1) [2..]
rationals = fix ((1:) . (>>= \x -> [x+1, 1%(x+1)])) :: [Rational]
powerset = filterM (const [True, False])
histogram = map (head &&& length) . group . sort
```

### 1.2.3. For Mathematicians

Every time someone writes `i = i + 1`, a mathematician dies[3]. The fact is that many mathematicians have cringed at the sight of a computer screen with some random code. They are used to writing stuff like:

> Let a function $f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = 2y + 3$, where $y = |x - 4|$. If we consider set $A = \{-5, -3, \ldots, 11\}$, we shall map function $f$ over $A$, naming the result set $B$. We shall also consider set $C = \left\{f^2(x) \,|\, x \in A, x < 10\right\}$.

One does not simply code such a thing in C or Python — at least not without mutilating maths. However, in Haskell, the result is pleasing to the eye and easy to understand, too (everything following the `--` is a comment).

```haskell
f :: Integer -> Integer
f x = 2*y + 3
    where y = abs (x-4)

a = [-5,-3..11] -- we'll see later why a, b, and c are lowercase
b = map f a
c = [(f x)^2 | x <- a, x < 10] -- this really works!
```

The mathematical applications of Haskell are endless. It's even possible to define and work with monoids [XREF]!

[3]Not really, but hey.

### 1.2.4. For Everybody Else

Intelligent and/or hardworking people will enjoy the challenge provided by Haskell. At the end of the journey, the traveller will look at the world with new eyes, satisfied that he is now better equipped to understand the Universe.

This is all because Haskell is riddled with complex, counterintuitive or simply mind-boggling elements. Let's take a look at something interesting.

```
compare 2 3    -- works
compare (2 3) -- doesn't work
(compare 2) 3 -- works!!
```

This "paradox" (let's call it *Problem Z* even though it's actually a feature), and more, will be presented and explained throughout the book.

#### Your turn! Exercises

These aren't exercises per se, but it's good to get used to the format. Head to the end of the book if you're really stuck and you need hints. Please don't overdo it. It's bad.

1. Why do *you* want to learn Haskell?
2. What other languages do you know[4]? Do you believe they will help or hinder your relationship with Haskell?
3. *Third exercise. This would be a harder one as indicated by the asterisk.
4. **This is an even harder exercise.

## 1.3. Before We Start

This book requires a Haskell interpreter. For most people, the best option is The Haskell Platform, although alternatives like hugs exist. The Haskell Platform uses GHCi as the interpreter (and also has a compiler, GHC), which is what we will use in our examples.

### 1.3.1. Using GHCi

On Windows, GHCi can be opened using the Start Menu. On Linux, Mac and other UNIX-like systems, ghci can be started using the shell. Below is a typical GHCi session on Linux. We type some expressions, load a file, add a module, and finally change the prompt to something shorter.

We added some blank lines to make the output more readable, but in real life the following is a single block of text. There's no need to understand it for now — the example is just to give a rough idea of the GHCi experience.

```
ee@bt:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.

Prelude> 2 + 3
5
Prelude> max 10 2
10
```

[4]If this is your first language, just like it was when I started out, congratulations! Haskell is a really nice language and it will grow on you. Enjoy the journey!

```

Prelude> :l test.hs -- loading a file
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.

*Main> import Control.Monad -- importing a module

*Main Control.Monad> :set prompt "ghci>␣"

ghci> :q -- you can also exit with Ctrl-D
Leaving GHCi.
ee@bt:~$
```

### 1.3.2. Interactive vs. Noninteractive

GHCi is very narrowly scoped. It's more of a debugger: you can't just copy-paste source files into it, like in Python — there are key differences between interactive code and code loaded from a file.

For example, compare the following (from now on we will use `ghci>` to indicate an interactive prompt — it's set using `:set prompt "ghci> "`) pieces of code. Both define `a` and `b` to be `5` and, respectively, `a + 1`. The first one is coded in a file and the second is written at the interactive prompt.

```
a = 5
b = a + 1
```

```
ghci> let a = 5
ghci> let b = a + 1
```

We will later (in [XREF]) understand why these differences occur. For now, remember that the second example is working *inside* a Haskell program (GHCi is, after all, written in Haskell).

### 1.3.3. Loading Files

Many examples will use functions written in a separate file, which is then loaded into GHCi. Let's go ahead and open up vim (or any other text editor) and write some declarations to get the hang of it.

```
-- File: basic.hs
a = 2
b = 3
c = a + b
```

Now let's load this into GHCi and see if it works (the file needs to be in the directory where GHCi was started, or it won't work[5]). `:l` stands for load, and in fact you can use `:load` instead.

```
ghci> :l basic.hs -- this is how we load files
[1 of 1] Compiling Main             ( basic.hs, interpreted )
Ok, modules loaded: Main.
ghci> a + 1
3
ghci> c - b == a
True
ghci> :r -- this reloads the file if we change it
```

[5]Unless you give it the full path to the file. For instance, `:l /home/ee/Code/Haskell/project/stuff.hs`

```
[1 of 1] Compiling Main             ( basic.hs, interpreted )
Ok, modules loaded: Main.
ghci>
```

Again, there is no need to dissect the above pieces of code — what's important is knowing how to load a file (`:l file.hs`) and reload it (`:r`).

## Your turn! Exercises

So now we have a basic idea of what to look for. We don't exactly know how to do a lot of stuff, so these exercises will be simple.

1. Install a Haskell compiler/interpreter if you haven't done so already, and open the interactive prompt.
2. Open up a text editor and create `starting-out.hs`. Write in Haskell-speak that `a` equals two, `b` equals three, `c` is `a` plus `b`, and `d` is the product of `a`, `b`, and `c`. Load the file you have just created.
3. Is `d` equal to 20? Go ahead and test it out using the interactive prompt.
4. `b` is now 5. Change the file to reflect the new reality and reload it.

# 2. Basics: Functions and Lists

I kinda expect functions to return something sensible, but I guess I'm spoiled by exposure to functional programming.

*(kzm)*

## 2.1. Getting Started

### 2.1.1. Simple Arithmetic

It is very easy to use GHCi as a calculator. It supports all the basic operations and some extra functions (`min`, `abs`, `exp` etc.). As an added bonus, Haskell supports arbitrarily large integers.

```
ghci> 4 + 5*6
34
ghci> exp 2
7.38905609893065
ghci> 10 - 4 - (max 5 6)
0
ghci> 10^60
1000000000000000000000000000000000000000000000000000000000000
```

There still are some problems, especially with the `-` operator.

```
ghci> -3
-3
ghci> -3 + 4
1
ghci> min -3 4 -- this gives a very long error message.
```

GHCi treats `min -3 4` as `min - (3 4)`, and therefore thinks we want to subtract `3 4` from `min`. This may look strange, even downright stupid, but GHCi has a very good reason: being able to call functions as arguments is essential in Haskell.

We have no choice but to oblige — a solution is to wrap `-3` in parentheses.

```
ghci> min (-3) 4
-3
```

### 2.1.2. Boolean Algebra

In Haskell, working with booleans or testing for equality is as straightforward as can be expected.

```
ghci> False || False -- right associative
False
ghci> True || False && False -- && has a higher precedence
True
```

```
ghci> not True
False
ghci> not False || not True
True
ghci> 5 == 6 -- by the way, equality is not associative
False
ghci> 5 /= 7 -- programmers beware, it's not !=
True
```

A combination of right associativity and something called *laziness* (we'll get back to it later) means that `||` stops at the first `True` statement found (from the left). Likewise, `&&` stops at the first `False`. Essentially, they stop because there's no point in continuing. `True || anything` is `True`, so why bother to see what that `anything` is? It doesn't matter.

Another interesting fact is that `||` and `&&` are not built into the language, they're functions like all others.

### 2.1.3. Calling and Making Functions

Functions are called with space between the parameters. Some functions accept only one parameter, some more[1]. We have already seen some functions, so here are some more examples, and then we'll move on.

```
ghci> succ 3 -- needs to have a logical successor
4
ghci> succ 'a'
'b'
ghci> pred 'Y' -- same here
'X'
ghci> pred "Hello" -- error
```

Before we do that, let's discuss why `"Hello"` doesn't have a predecessor. One might think that it's `"Helln"` but that is not the case. In Haskell, as in most languages out there, `"a" < "aa" < "aaa" < "ab" < "b"`. You can always find a string that is "closer" to `"hello"` than the one you've just found.[2]

There is an important distinction to be made regarding function calls. Parentheses around the arguments only set precedence, not separate the function from the arguments. It's essential not to get fooled, especially in the next example.

```
ghci> foo (bar 10) -- in C this would be foo(bar(10))

ghci> (foo bar) 10
ghci> foo bar 10 -- this is equivalent to the above

ghci> foo bar (baz 10) 8 -- in C: foo(bar, baz(10), 8)
```

Also, function application has the highest precedence, so if you write `foo 10 + 8`, it means `(foo 10) + 8` (for more details see A.1.1).

We're slightly familiar with defining functions, too (the 1.2.3 example). Let's play a little more with them. Obviously, we can refer to other functions in a definition. Another thing to note is that functions can't begin with uppercase letters.

[1]Technically all functions accept only one parameter, but it's not healthy to think like this, at least for now — remember *Problem Z* (introduced in 1.2.4)?

[2]The same argument can be made for rational numbers, i.e. "what is the predecessor of 1.2". Haskell has a somewhat non-mathematical way of dealing with predecessors of non-natural numbers, because of the way they're internally defined. [FIXME-ranges] [XREF]

```haskell
-- File: functions.hs
triple x = 3*x
strangeAddition x y = x + triple y
squareTwo x y = (x + y)^2

c = 4 -- this one takes zero parameters
```

Before we start... calling around, let's talk a little about the last line. This is a very interesting case indeed — `c` is what we would call in other languages a "variable". It's declared the same as a function, but it takes zero parameters so it's a constant[3] (that's why Haskell gives an error if you do `c = 4` then `c = 5` in the same file).

Unlike most languages, in Haskell a zero-parameter function and a constant are really the same. This, strangely enough, has something to do with *Problem Z* — we'll understand what that means soon enough.

```
ghci> :l functions.hs
[1 of 1] Compiling Main             ( functions.hs, interpreted )
Ok, modules loaded: Main.
ghci> triple 2
6
ghci> strangeAddition 10 20
70
ghci> squareTwo 5 6
121
ghci> triple c
12
ghci> strangeAddition (triple 2) c
18
```

Before we continue, let's look a bit at Haskell's if-else. The first thing we notice is that the `else` part is mandatory. Why? Every function has to return something. Why? Haskell is more like maths — there are no "variables" to change, so a function that doesn't return anything wouldn't work[4]. Does "$f(x) =$" make sense?

Let's add something to `functions.hs` (the quote is a valid character in function names) and see what happens. Indentation is essential in Haskell because that's how the interpreter identifies blocks of code. This is pretty much self-explanatory. If the statement after the `if` is true, then it evaluates the `then` part, else it evenuates the `else` part.

```haskell
-- File: functions.hs (CONTINUED)
strangeAddition' x y = if x > y
                       then x + triple y
                       else y + triple x
```

```
ghci> :r -- we won't be showing load/reload from now on
[1 of 1] Compiling Main             ( functions.hs, interpreted )
Ok, modules loaded: Main.
ghci> strangeAddition 5 3
14
ghci> strangeAddition 3 5
18
ghci> strangeAddition' 5 3
14
```

[3]Mathematicians will understand this right away.
[4]There is also a technical reason, explained in detail in [XREF]

```
ghci> strangeAddition' 3 5
14
```

### 2.1.4. Infix Functions

Until now we've called functions by putting them before the arguments, like above. But if we surround functions with backquotes, we can make them infix (put them between the parameters), much like `+` or `*`.

***Warning!*** Backquotes work only with two-parameter functions.

```
ghci> 3 `squareTwo` 4
49
ghci> 10 `strangeAddition` 20
70
ghci> 2 `triple` -- error (and looks stupid, too)
```

Backquotes are usually adopted to make functions more readable, but they can also be used to create chains. Watch out for associativity (default left) and precedence (order of operations, by default highest) — built-in functions don't use the defaults (see A.1.1).

```
ghci> 2 `squareTwo` 3 `squareTwo` 4 `squareTwo` 5
715716
ghci> ((2 `squareTwo` 3) `squareTwo` 4) `squareTwo` 5
715716
ghci> 2 `squareTwo` (3 `squareTwo` (4 `squareTwo` 5))
49815364
```

If a function name contains only symbols (like `++`, `^`, or `-.-`), it's automatically infix. We can still call infix functions before the arguments, by putting them in parentheses. This really helps with *Problem Z*.

```
ghci> (+) 2 3
5
ghci> (*) 4 5
20
ghci> (/) 10 4
2.5
```

### Your turn! Exercises

We're now somewhat familliar with basic math in the interpreter and we can do a handful of things with functions. Let's consolidate this knowledge with a couple of easy questions and a few more advanced ones.

1. Fire up GHCi and try the following calculation: $\frac{\frac{1+2}{4}+\frac{5}{3+6}}{\frac{7}{8+9}+\frac{1}{10}}$. Do it in a single line (no intermediate steps). What do you notice? How easy would it be for someone else to understand what you wrote?
2. Calculate the maximum between 2, 3, and 5. Now do it without using any parentheses (on a single line). Can you do it using `max` only once?
3. Create and load a file (use whatever name suits you) that contains: a function that calculates the maximum between three numbers, a function that multiplies three numbers, a function that adds three numbers, and a function that checks if three numbers are equal.
4. *Write a function that calculates the maximum between two numbers. You aren't allowed to use `max`.[5] Do it in two different ways.

[5]You aren't allowed to use `min` either, but bonus points if you thought of it!

## 2.2. Using Lists

### 2.2.1. Intro

Lists are to Haskell like... well, there's really no comparison. They are the most used data structure. They:

- Are homogenous — mixing, for example, numbers with characters gives an error.
- Have variable length[6].
- Can be infinitely long[7].
- Are singly linked — lists can only be traversed from left to right[8].

We'll define some lists in a file so we can explore functions that operate on them.

```haskell
-- File: lists.hs
numbers = [1, 3, 7, 5, 6, 6, 8, 10]
languages = ["lisp", "haskell", "c", "perl", "ruby", "python"]
hello = "Hello,␣World!" -- same as ['H', 'e', 'l', 'l', ...and so on]
listOfLists = [[1, 5, 7, 9], [2, 4, 6], [1]]
emptyList = []
```

For starters, ++ concatenates two lists. It's one of the most basic operators. It's associative, so `(a ++ b) ++ c` is equivalent to `a ++ (b ++ c)`[9].

```
ghci> [1, 2, 3] ++ [5, 4]
[1,2,3,5,4]
ghci> "Haskell" ++ "␣" ++ "is" ++ "␣" ++ "fun"
"Haskell␣is␣fun"
```

The simplest list operator is `:` — it adds an element to the front of a list[10]. It's so basic, in fact, that `[1, 2, 3]` is just syntactic sugar[11] for `1:2:3:[]`. In 4.1.3 and [XREF] we'll cover the many uses of `:`, but for now we'll stick to basics.

```
ghci> 5 : [4, 6, 8]
[5,4,6,8]
ghci> 5 : 4 : 6 : 8 : []
[5,4,6,8]
ghci> 'f' : "iretruck"
"firetruck"
ghci> [3, 4] : [[5, 6, 7], [8, 9]]
[[3,4],[5,6,7],[8,9]]
```

The following throw errors because we're not using `:` correctly. There are numerous ways to fix them, however.

```
ghci> [1] : [2, 3]    -- use 1 : [2, 3]  or [1] ++ [2, 3] instead.
ghci> 1 : 2 : 3       -- use 1 : 2 : [3] or 1 : 2 : 3 : []
ghci> [10, 9, 2] : 4 -- use [10, 9, 2] ++ [4]
```

[6]Well, technically speaking they can't change (nothing can), but for all intents and purposes they are "variable" in length.
[7]This is because of *laziness*. Functions in Haskell (like those from 2.1.2) are made to use only as much information as is necessary, and not more. If we combine with `&&` an infinite number of `False`s, do we really need to get past the first one?
[8]This means that accessing the last element requires going through the whole list — watch out!
[9]Without this basic property, lists would be stupid.
[10]`:` is called a list constructor (or cons for short). It's the operator that "links" the elements of a list (we'll see how this happens a bit later, in [XREF])
[11]The same thing, but prettier.

### 2.2.2. Basic List Functions

Getting information from lists is done using the following built-in functions (we usually call our lists `xs`[12]):

- `head` — first element
- `tail` — all but the first
- `last` — last element
- `init` — all but the last
- `!! n` — the nth element (numbering starts at 0)
- `take n` — first n elements
- `drop n` — all but the first n elements
- `length` — self-explanatory
- `null` — check if the list is empty. How *not* to do it:
  - `list == []` — bad
  - `length list == 0` — worse
  - `unsafeCoerce list :: Bool` — worst

```
ghci> let xs = [1, 2, 3, 4, 5, 6]
ghci> head xs
1
ghci> tail xs
[2,3,4,5,6]
ghci> last xs
6
ghci> init xs
[1,2,3,4,5]
ghci> xs !! 4
5
ghci> take 2 xs
[1,2]
ghci> drop 2 xs
[3,4,5,6]
ghci> length xs
6
ghci> null xs
False
```

One thing worth pointing out is that, due to the nature of lists in Haskell, accessing the last element of a list is considerably slower than accessing the first one. This is because, internally, accessing an element requires "going through"[13] the ones before it. [FIXME-elaborate with examples]

***Warning!*** Giving out-of-bounds values to `head`, `tail`, `init`, `last`, and `!!` throws an exception.

```
ghci> head []
*** Exception: Prelude.head: empty list
ghci> l !! 100
*** Exception: Prelude.(!!): index too large
ghci> l !! (-2)
*** Exception: Prelude.(!!): negative index
```

[12]As in the plural form of `x` — "exes". Along the same lines: `ys`, `zs`, `as`, `bs`, `cs` etc.
[13]This is not entirely accurate, but it will do for now.

Some more useful functions:

- `maximum` — the maximum of a list[14]
- `minimum` — the minimum
- `sum` — the sum of a list of numbers
- `product` — likewise, the product
- `elem` — checks if an element is a member of a list[15] (usually called infix because it's more readable)
- `notElem` — the opposite of `elem` (also called infix).

```
ghci> let xs = [8, 5, 3, 4, 10, 2]
ghci> maximum xs
10
ghci> minimum xs
2
ghci> sum xs
32
ghci> product xs
9600
ghci> 5 `elem` xs
True
ghci> 22 `elem` xs
False
ghci> 22 `notElem` xs
True
```

A special case, `concat`, operates on lists of lists: it flattens them. It only "removes" one layer, though.

```
ghci> concat [[2,3],[4,5]]
[2,3,4,5]
ghci> concat [[5]]
[5]
ghci> concat [[[5]]]
[[5]]
```

There are some functions that operate on lists of `Bool`s:

- `and` — returns `True` if all the elements are `True`, `False` otherwise.
- `or` — `True` if at least one is `True`, `False` otherwise.

```
ghci> and [True, True, False]
False
ghci> and [True, True, True]
True
ghci> or [True, False, False]
True
ghci> or [False, False, False]
False
```

And neither last nor least (see C.1 for more), `reverse` reverses a list. It's not very efficient, though, so avoid reversing long lists.

[14]To calculate the maximum, the elements need to have some sort of logical order. A list of numbers or a list of characters are fine, but a list of functions is not.

[15]Needs to be able to equate elements. This may seem pretty standard, but not all stuff can equal other stuff (we'll discuss this in-depth in [XREF]).

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

### 2.2.3. Ranges

Many times we need to construct lists according to certain rules. Probably the simplest way is by using ranges. Let's see some examples and then discuss them.

```
ghci> [1, 2 .. 20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> [1 .. 20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> [1, 3 .. 15]
[1,3,5,7,9,11,13,15]
ghci> [1, 7 .. 30]
[1,7,13,19,25]
ghci> [3, 2 .. -10]
[3,2,1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

The following will *not* work.

```
ghci> [1, 2, 4, 8 .. 128] -- nope
ghci> [1 .. 39, 40] -- not this, either
```

It's pretty obvious: these ranges generate sequences where the difference between consecutive terms is constant (arithmetic progressions).

They always go like this: `[first element, next element .. last element]`.

If we need to generate consecutive things, `[a .. n]` is shorthand for `[a, a+1 .. n]` which is shorter than writing the whole list by hand.

Furthermore, only arithmetic progressions are possible using ranges. You can, however, specify any step, including negative or noninteger[16] ones.

```
ghci> [1, 2.1 .. 5]
[1.0,2.1,3.2,4.300000000000001,5.400000000000001]
```

***Warning!*** Using nonintegers in ranges yields undesireable results due to rounding errors.

Interestingly, if the upper bound is omitted, ranges generate infinite lists, as exemplified below[17]. If you do this, press `Ctrl-C` to stop it.

```
ghci> [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
    22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
    58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
    76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
    94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, ^CInterrupted.
```

How is this useful? Well, let's remember that Haskell is *lazy*, so unless we want something unwise, like printing all the elements of an infinite list (see above) we should be in the clear. We are already familiar with `take`, so let's use it in conjunction with ranges.

[16] With decimals.
[17] Disclaimer: we won't actually print infinitely many numbers.

```
ghci> take 20 [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> take 5 [13, 26 ..]
[13,26,39,52,65]
ghci> take 11 [1, -2 ..]
[1,-2,-5,-8,-11,-14,-17,-20,-23,-26,-29]
```

We immediately notice that the computations have ended, so clearly Haskell didn't evaluate the entire infinite list. In fact, when we learn more about functions, we'll see exactly how laziness works[18].

Also, take note: ranges aren't limited to numbers.

### 2.2.4. Cycling Lists

What if we want a number repeated over and over? We can do `[1, 1 .. ]`, and that's perfectly okay. There are three functions we have omitted from 2.2.2, and they will make it more readable. Additionally, they have the advantage of being functions, which will help with *Problem Z*. Here they are:

- `repeat` repeats an element into an infinite list. We'll probably want to `take` a finite number of elements, though.
- `cycle` repeats an entire list. Again, we'll want to `take` elements.
- `replicate` repeats an element a specified number of times.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
ghci> take 10 (cycle [5, 4])
[5,4,5,4,5,4,5,4,5,4]
ghci> replicate 10 4
[4,4,4,4,4,4,4,4,4,4]
```

***Warning!*** Do not confuse `repeat` and `cycle` — they do very different things.

```
ghci> take 10 (repeat [5, 4])
[[5,4],[5,4],[5,4],[5,4],[5,4],[5,4],[5,4],[5,4],[5,4],[5,4]]
ghci> take 10 (cycle [5, 4])
[5,4,5,4,5,4,5,4,5,4]
```

## 2.3. List Comprehensions

### 2.3.1. Basics

We've seen how to declare, manipulate and, to an extent, generate lists. We will now learn one of the most powerful tools in all of Haskell, list comprehensions. Let's start with basic examples and move on from there.

```
ghci> [ x | x <- [1..20] ]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> [ x | x <- [1..20], even x ]
[2,4,6,8,10,12,14,16,18,20]
ghci> [ x | x <- [1..20], x > 6 ]
[7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

[18]It's not unlike if-else in other languages — if the statement is true, the `else` branch won't evaluate and viceversa.

```
ghci> [ x | x <- [1..20], even x, x > 6 ]
[8,10,12,14,16,18,20]
ghci> [ x | x <- [1..20], even x, x > 6, odd x ]
[]
ghci> [ a ++ b | a <- ["Haskell␣", "C␣"], b <- ["syntax", "types"] ]
["Haskell␣syntax","Haskell␣types","C␣syntax","C␣types"]
ghci> [ x + 3 | x <- [1, 6 .. 30] ]
[4,9,14,19,24,29]
ghci> [ x + 3 | x <- [1, 6 .. 30], even x ]
[9,19,29]
ghci> [ a ++ "␣is␣fun!" | a <- ["Haskell", "Perl", "C", "Lisp"] ]
["Haskell␣is␣fun!","Perl␣is␣fun!","C␣is␣fun!","Lisp␣is␣fun!"]
```

Anyone who's seen and understood mathematical set comprehensions can just skim the rest of the section. 2.3.2 is worth reading carefully, though.

List comprehensions have two components (let's take `[ 2*x | x <- [1, 3, 4], odd x ]` as an example):

- The left hand-side contains the expression to be evaluated (in our case, `2*x`)
- The right hand-side has:
  - A base list from which `x` is extracted: `x <- [1, 3, 4]`
  - A list of predicates (filters) that must be satisfied (in this case, we have only one): `odd x`

In order to understand better, let's manually calculate the above comprehension, step by step.

1. Find the base list: `[1, 3, 4]`.
2. Take the first element from the base list and call it `x`.
3. Check the truth value of the predicates (in this case, only one): `odd x`.
4. If *all* the predicates are satisfied, evaluate the left hand-side expression for `x`: `2*x` then add it to the result list.
5. Do the above steps for all elements in the base list.

Voilà: the result is `[2, 6]`. It's important to note that internally, Haskell does things a little differently. However, the result is the same so it shouldn't bother us.

### 2.3.2. Advanced Uses

We can also combine two, three or more base lists, more predicates etc. The order of the base lists determines the order of the result list, as we can see from the first example. The predicates are calculated left-to-right so it's recommended that more powerful filters be put first.

```
ghci> [ 10*a + b | a <- [1..3], b <- [1..3] ]
[11,12,13,21,22,23,31,32,33]
ghci> [ x * y | x <- [2, 4, 6], y <- [10, 100, 1000] ]
[20,200,2000,40,400,4000,60,600,6000]
ghci> [ x * y | x <- [1..4], y <- [1..3], even (x + y) ]
[1,3,4,3,9,8]
ghci> [ x + y | x <- [3..6], y <- [2, 4, 8], x <= y ]
[7,11,8,12,13,14]
```

Because a list comprehension is an expression, we can put it in the left hand-side of another one — comprehensions inside comprehensions.

```
ghci> let xss = [[1, 2, 3, 4, 5], [4, 5, 6, 7], [7, 8, 9, 10]]
ghci> [ [ x | x <- xs, x >= 5 ] | xs <- xss ]
[[5],[5,6,7],[7,8,9,10]]
```

Moreover, instead of specifying an upper bound in a base list, we can `take` a number of results afterwards.

```
ghci> take 5 [ a | a <- [1..], b <- [1..a], c <- [1..b], a^2 == b^2 + c^2 ]
[5,10,13,15,17]
```

There are a few catches, however, some very serious.

```
ghci> take 20 [ x | x <- [1..], x < 10 ]
[1,2,3,4,5,6,7,8,9^CInterrupted -- this would never finish
ghci> take 5 [ x | x <- [1..], x < 10 ]
[1,2,3,4,5] -- this works fine because Haskell is lazy
```

***Warning!*** Make sure Haskell can find at least as many items as you `take`.

Some problems are harder to spot without running the code. For instance, Haskell never tries `x = 2` in the following example, because it has plenty of `y`s to choose from.

```
ghci> take 20 [ x * y | x <- [1..], y <- [1..] ]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

To repeat, Haskell tries all the values from the *last* base list before continuing, so avoid having more than one unbounded base list, because it will either not give us what we want (see above) or run indefinitely[19] (see below). Actually, there is a mountain of theory on this issue, such as this paper (advanced content).

```
ghci> take 10 [ x * y | x <- [1..], y <- [1..], y <= x ]
[1^CInterrupted. -- bad idea, runs indefinitely
ghci> take 10 [ x * y | x <- [1..], y <- [1..x] ]
[1,2,4,3,6,9,4,8,12,16] -- do this instead
```

Mastering all the subtleties of list comprehensions takes a lot of time and experience, so let's move on. We'll learn as we go.

### 2.3.3. Practical Applications

On the up side, list comprehensions have many practical uses. The classical example is determining the length of a list. We'll need to apply our knowledge of list functions here, namely `sum`.

```
ghci> sum [ 1 | x <- [3 .. 20] ]
18
```

It works, but we're not really using `x` anywhere, so it's a waste of a perfectly good variable name. The solution is to write an underscore whenever a variable name is not needed.

```
ghci> sum [ 1 | _ <- [3 .. 20] ]
18
```

If we want to use them repeatedly, we can declare functions with list comprehensions. Some examples:

[19]This is called a diverging computation.

```haskell
-- File: comprefunctions.hs
length' xs = sum [ 1 | _ <- xs ]
vowels string = [ c | c <- string, c `elem` "aeiou" ]
removeVowels string = [ c | c <- string, c `notElem` "aeiou" ]
allSums xs ys = [ x + y | x <- xs, y <- ys ]
```

```
ghci> length' [2, 4 .. 10]
5
ghci> length' []
0
ghci> vowels "hello␣world"
"eoo"
ghci> removeVowels "hello␣world"
"hll␣wrld"
ghci> allSums [1, 2, 3] [4, 5]
[5,6,6,7,7,8]
```

Functions and lists have a lot of power. We'll be using them extensively throughout this book (and even outside it) so it's better to take our time and make sure we understand as much as we can at this point. Things are only going to get harder as we advance.

# 3. Types, Typeclasses, and Polymorphism

I should actually think before coding, but the type system is so good :)

*(Cale)*

## 3.1. Understanding Types

### 3.1.1. Knowing Types

In most of the programming world, every variable has a type: an integer, a character, a boolean etc. But more often than not, they're there for cosmetic purposes — most compilers will happily add a number to a character. That doesn't make much sense, does it?[1]

Fortunately, Haskell has a strong type system. That means that however similar their internal representations are, the compiler won't allow us to perform illogical calculations on them, such as multiplying an integer with a boolean. This may seem restrictive (and it sometimes is), but it helps avoid certain types of errors[2] (type errors).

Moreover, Haskell features static typing, which means all types are known at compile-time so if the program has a type error, it won't even compile.

As an added bonus, Haskell has type inference, so we don't need to manually specify the type of everything we use. Basically, the compiler can figure out on its own that `1` is a number or `"hello"` is a string[3].

In GHCi, we can use `:t` to determine the type of an expression (`::` means "has the type of").

```
ghci> :t 'a'
'x' :: Char
ghci> :t "abcd" -- same as ['a','b','c','d']
"xxx" :: [Char]
ghci> :t 'a':'b':'c':'d':[] -- same as "abcd"
'a':'b':'c':'d':[] :: [Char]

ghci> :t False
False :: Bool
ghci> :t "hello" == "world" -- returns False
"hello" == "world" :: Bool
```

We know that `[]` denotes a list, so it's easy to conclude that `[Char]` means a list of characters. The others are self-explanatory. This is just a very short example — we'll be seeing more in the future.

We also immediately notice that all types begin with a capital letter. This is the reason why variable and function names are lowercase[4].

Below is a recap of the most widely used types in Haskell. We'll be running into these all the time.

[1] One might argue that `'z'` is `'a' + 25`, but Haskell won't let you do that.

[2] Imagine working on a long, difficult physics problem asking for some velocity — but after hours of calculations, the result is in kilograms. That can't be good.

[3] It can even deduce more complex types just as easily.

[4] The capitalization technique used for functions in Haskell is informally named `camelCase`.

- `Int` is a bounded integer. On 32-bit systems it's between $-2^{31}$ and $2^{31}-1$.
- `Integer` is an arbitrarily large integer. It's slightly less efficient than `Int`.
- `Float` is a single-precision floating point.
- `Double` is a double-precision floating point. Due to optimizations, `Double` can be faster than `Float`.
- `Bool` is a boolean. It can be either `True` or `False`. `1` and `0` won't work.
- `Char` represents (by default) a Unicode character.

If we try to mix wrong types, Haskell throws a type error. It usually looks like this:

```
ghci> 3 + 'a'

<interactive>:1:1:
    No instance for (Num Char)
      arising from the literal `3'
    Possible fix: add an instance declaration for (Num Char)
    In the first argument of `(+)', namely `3'
    In the expression: 3 + 'a'
    In an equation for `it': it = 3 + 'a'
```

Basically GHCi tells us that it doesn't know how to add `'a'` to `3`, because `'a'` is not a number. An extremely detailed dissection of type errors in GHCi is presented in B.2.1.

### 3.1.2. Type Declarations

In Haskell, functions have types too. We mentioned that Haskell can infer the type of an expression on its own. However, it's possible to manually declare the type of a function. This helps us to:

- Clarify our thoughts
- Make code more readable
- Avoid mistakes

The type declarations make functions much more expressive. Although Haskell could have inferred by itself what the types of the functions are (like in the 2.1.3 and 2.3.3 examples), we chose to give explicit type declarations to illustrate the method.

In type declarations the parameters (and the return type) are separated by `->`, regardless of how many of them there are[5].

```haskell
-- File: functions2.hs
triple :: Int -> Int
triple x = 3 * x

strangeAddition :: Int -> Int -> Int
strangeAddition x y = x + triple y

squareTwo :: Double -> Double -> Double
squareTwo x y = (x + y)^2

vowels :: [Char] -> [Char]
vowels word = [ c | c <- word, c `elem` "aeiou" ]

sumLists :: [Int] -> [Int] -> [Int]
sumLists xs ys = [ x + y | x <- xs, y <- ys ]
```

[5] *Problem Z* is at work here. We'll see why it's not something like `Int, Int -> Int`.

***Warning!*** The parameters and the return type are not differentiated — all are separated by `->`.

In fact, type declarations give us so much information, that we can even deduce what a function does simply from its type declaration.

Let's take `f :: [Char] -> Int` as our example. This function takes a list of characters (a string) and returns an integer. We can reasonably infer that the function takes the string and performs some sort of counting (such as finding out the total length or counting all the spaces) or other calculation (such as a hash function).

Indeed, `f` is defined like so: `f xs = [ 1 | x <- xs, x ‘elem‘ "abc" ]`. The function counts all occurences of the letters `a`, `b`, and `c` in a given string, so our assessment was spot-on.

Because of this tremendous advantage, we'll be giving type declarations to (almost) every function we write from now on.

Oh, and just so we don't forget. If we have two functions with the same type declarations, we don't need to repeat ourselves — we separate the function names with commas in their type declaration.

```haskell
-- File: functions2.hs (CONTINUED)

sum1, sum2 :: Int -> Int -> Int -> Int
sum1 x y z = x + y + z
sum2 x y z = x + y - z
```

## 3.2. Polymorphism

### 3.2.1. Type Variables

Until now, we've defined functions of type `Int -> Int` or `[Char] -> Int`. But what about functions like `head`? If we give `head` a type declaration of `[Int] -> Int`, for example, it will work only with integers. But `head` works with basically every type of element. So what is `head`'s type?

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

In the above snippet of code, `a` is what we call a *type variable*. It's some sort of generic type. Because `head` doesn't require specific behavior out of its parameters (unlike `==`, for instance, which requires parameters that can be equated), we can use `a`[6] to make an extremely general function. Basically `[a] -> a` tells us that it accepts a list of any type and returns an element of *the same* type.

This is called *polymorphism*: whenever we use a type variable, we indicate that the function does not expect a specific behavior, so it basically works as-is for a variety of inputs.

### 3.2.2. Typeclasses

We've seen some of the most specific type signatures (like `Int -> Int` or `Char -> Int -> Bool`) and the most general (for example, `[a] -> a`, `[a] -> [a] -> [a]`), but what if we require something in between? For this, we need typeclasses.

Typeclasses "group" types with a common behavior. Each internal definition of a typeclass contains a collection of functions that must work for all members of that typeclass. It's pretty simple really.

Typeclasses are presented in depth in B.1 (strongly recommended reading). In the following we'll try to explain how they *interact*. For this, we'll consider `Num` and `Integral`. `Num` "contains" all types of numbers, but `Integral` only integers.

[6]It doesn't need to have only one letter, but for conciseness, we'll use `a`, `b`, `c` etc.

In addition, for something to be an `Integral`, it must also be a `Num`. We can logically conclude that `Integral` is some sort of a *subclass*[7] of `Num`: it is more “specific”. The more specific a typeclass, the more operations are possible within it. For example, `Num` supplies, among others, `+`, `-`, `*` and `abs`. `Integral` offers, in addition, things like `div` (integer division; `/` in other languages) and `mod` (modulo; `%` in other languages).

If we just write `20` or `30`, they’re any type[8] of numbers. But as soon as we perform an `Integral` specific function on them, they (and the result of the operation) can no longer be `Float`s or `Rational`s or whatever. We’ll get round to `=>` in a few moments.

```
ghci> :t 20
20 :: Num a => a
ghci> :t 30
30 :: Num a => a
ghci> :t 20 `div` 5
20 `div` 5 :: Integral a => a
ghci> :t 20 `mod` 30
20 `mod` 30 :: Integral a => a
```

This is the gist of typeclasses and polymorphism: they group common behavior so we can make very general functions. If we make a `sort` function, we can be certain that it won’t only work with lists of numbers, but also with strings or anything else that can be ordered.

At this point, it’s a good idea to go through the typeclasses described in B.1. They’re very useful.

### 3.2.3. Making Polymorphic Functions

Now let’s see how we actually use typeclasses: in type declarations, mostly. Here are a few examples:

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t (^)
(^) :: (Num a, Integral b) => a -> b -> a
ghci> :t pi
pi :: Floating a => a
ghci> :t show
show :: Show a => a -> String
```

It seems polymorphic functions really do use the `=>` a lot. Basically, everything before the `=>` is a class constraint. In the first example, it tells the compiler (and us) that `a` is a member of `Num`[9]. The actual type of the function is right after the `=>`.

When we read such a definition, we usually do it (somewhat) from right to left.

We shall use `(^) :: (Num a, Integral b) => a -> b -> a` as an example.

- `(^)` is the name of the function. In this case it’s surrounded by parentheses because it consists only of symbols.
- `::` means “has type of” — now we jump to the bit after the `=>`.
- `a -> b -> a` means the function takes a parameter of a type (`a`), a parameter of another type (`b`) and returns a parameter of the first type (`a`).

[7]Calling it a subclass is not technically correct, but it *is* intuitively true.
[8]We’ve avoided using “kind” to the point of repeating ourselves. This is not due to lack of vocabulary: in Haskell, *kind* means something different. Kinds are explained in [XREF] (advanced topic).
[9]We can also have multiple class constraints by surrounding them in parentheses and separating them with commas, like in `(^)`.

- `(Num a, Integral b)` is the last thing we read — it tells us that `a` is any type of number but `b` is an integer[10].

Now we'll apply our newly-gained knowledge to make our functions more general. We'll recycle examples from 2.1.3, 2.3.3, and 3.1.2.

```haskell
-- File: polyfunctions.hs

triple :: Num a => a -> a
triple x = 3*x

strangeAddition :: Num a => a -> a -> a
strangeAddition x y = x + triple y

c :: Num a => a
c = 4

length' :: Num a => [b] -> a
length' xs = sum [1 | _ <- xs ]

vowels :: [Char] -> [Char]
vowels word = [ c | c <- word, c `elem` "aeiou" ]

sumLists :: Num a => [a] -> [a] -> [a]
sumLists xs ys = [ x + y | x <- xs, y <- ys ]
```

A great thing about Haskell is that if our type definitions are wrong (i.e., they are incompatible with the function itself), an error is thrown. Apart from the obvious advantage, this means we can "cheat" and let Haskell infer the type for us, then copy-paste it in our file.

```
ghci> let spaces xs = sum [ 1 | x <- xs, x == ' ' ]
ghci> :t spaces
spaces :: Num a => [Char] -> a
```

```haskell
-- File: polyfunctions.hs (CONTINUED)

spaces :: Num a => [Char] -> a
spaces xs = sum [ 1 | x <- xs, x == ' ' ]
```

### 3.2.4. Drawbacks

We've seen how we can make our programs more readable and reliable by adding type definitions. The good news is that we can't accidentally add centimeters and inches. The bad news is that we can't add an integer and a floating point. *What?*

Of course we can do stuff like `4 + 5.1`, but that's different. Let's see.

```
ghci> 4 + 5.1
9.1
ghci> (4 :: Int) + (5.1 :: Float)

<interactive>:1:15:
    Couldn't match expected type `Int' with actual type `Float'
```

[10]It can be any one of the 7 types of integer Haskell has.

```
    In the second argument of ‘(+)’, namely ‘(5.1 :: Float)’
    In the expression: (4 :: Int) + (5.1 :: Float)
    In an equation for ‘it’: it = (4 :: Int) + (5.1 :: Float)
```

It seems that it all blows up if we force the types. The above error tells us, quite clearly, that it expected `5.1` to be an `Int` rather than a `Float`. Haskell can't add two different types[11]. The keen reader will remember that we previously mentioned polymorphic constants. We can easily check if this is the case here.

```
ghci> :t 4
4 :: Num a => a
ghci> :t 5.1
5.1 :: Fractional a => a
ghci> :t (4 + 5.1)
(4 + 5.1) :: Fractional a => a
```

Aha! So `4` can take any number type (`Int`, `Complex`, `Rational`, `Float`, `Double` etc.), but `5.1` is a fractional (`Float`, `Double` etc.). Naturally, adding them means that `4` can have only the types `5.1` can have, so anything in `Fractional`[12].

Right now, things may seem confusing (and rightfully so). The most important thing to remember here is to make type declarations as general as possible, but not more general. In bullet points:

- Specific declarations limit a function to a certain type or typeclass: `triple :: Int -> Int`.
- General declarations make a function versatile[13]: `triple :: Num a => a -> a`.
- Too general declarations are incorrect and throw errors: `triple :: a -> a`.

If we're not sure of a type, we should leave it blank. The compiler always infers types better than the user[14].

Some food for thought: what happens if a typeclass has the same name as a type? So, for example, we have `sillyFunction :: Derp a => a -> Derp`. How do we distinguish between the first `Derp` and the second one? Well, they're logically different: one is a type, the other a typeclass. It doesn't matter if both have the same name. Does anyone ever confuse Jack the actor with Jack the movie character[15]? In technical terms, we say that they have different *kinds* (we'll talk more about them in [XREF]). The compiler won't ever confuse them, and as it happens, it's a pretty frequently used technique: we don't want to... "pollute the namespace".

## 3.3. Case Study: Tuples

### 3.3.1. Lists Recap

We mentioned lists are homogenous and have variable length (2.2.1). Before continuing, let's explore this from a new perspective: types.

```
ghci> :t [1, 2, 3]
[1, 2, 3] :: Num t => [t]
ghci> :t [1, 2, 3, 4]
[1, 2, 3, 4] :: Num t => [t]
ghci> :t (:)
```

[11]The addition operator (+) is of the type `Num a => a -> a -> a`.
[12]Actually, it should look like `(4 + 5.1) :: (Num a, Fractional a) => a`, but because `Fractional` is "included" in `Num`, it's the same thing.
[13]Sometimes we want to avoid that. For example, maybe we want a function that can only triple integers so we don't accidentally rounding errors.
[14]Unless, of course, it's released software — type definitions are half the documentation.
[15]Or for physicists, $a$ the length with $a$ the acceleration.

```
(:) :: a -> [a] -> [a]
ghci> :t (++)
(++) :: [a] -> [a] -> [a]
```

Even if we don't know anything about lists, from the above piece of code we can draw two very important conclusions:

- No matter how long a list is, its type is the same. This makes them essentially "variable" in length — we have "do-it-all" functions that can lengthen (`:`, `++` etc.) or shorten (`take`, `drop` etc.) any list, regardless of length.
- Both `:` and `++` take identical types as parameters, so there's no way we can get away with adding a different type of element to a list.

This translates into our current knowledge of lists: variable length and homogeneity. It reinforces the idea that we can learn a great deal simply by analyzing types.

### 3.3.2. Understanding Tuples

Let's say we heard of a new Haskell feature: we can put stuff in parentheses and surround them by commas — these structures are called tuples[16]. Unfortunately all the documentation is lost (yeah, right). It may not seem like a lot, but we can extract a wealth of information from the little we know.

First, let's see if we got the syntax right and try various things to see if they work.

```
ghci> (4, 5, 6)
(4,5,6)
ghci> (10, 2, 3, 3)
(10,2,3,3)
ghci> (85, "Hello")
(85,"Hello")
ghci> ('a', "Haskell", 15, "never", "easy")
('a',"Haskell",15,"never","easy")
ghci> ()
()
ghci> ('a')
'a'
ghci> (20)
20
```

Let's draw some partial conclusions about tuples:

- They can be any size.
- They are *not* necessarily homogenous.
- There is such a thing as an empty tuple: `()`.
- Single-element tuples are the same as the elements themselves[17].

Let's see what types they are.

```
ghci> :t (4, 5, 6)
(4, 5, 6) :: (Num t1, Num t2, Num t) => (t, t1, t2)
ghci> :t (10, 2, 3, 3)
(10, 2, 3, 3) :: (Num t1, Num t3, Num t2, Num t) => (t, t1, t2, t3)
ghci> :t (85, "Hello")
```

[16]For the record, that's not a new feature.
[17]That's pretty obvious — all we did is surround them with parentheses.

```
(85, "Hello") :: Num t => (t, [Char])
ghci> :t ('a', "Haskell", 15, "never", "easy")
('a', "Haskell", 15, "never", "easy")
  :: Num t => (Char, [Char], t, [Char], [Char])
ghci> :t ()
() :: ()
ghci> :t ('a')
('a') :: Char
ghci> :t (20)
(20) :: Num a => a
```

So the type of the tuple contains the types of all the elements inside it. This means:

- Tuples have an essentially fixed length[18].
- An empty tuple is its own type: `()` is of type `()`.

We've also inadvertently learned that type definitions can be split across multiple lines (as long as the next lines are indented slightly to the right).

### 3.3.3. Functions on Tuples

We now make a horrible typo:

```
ghci> (,)

<interactive>:1:1:
    No instance for (Show (a0 -> b0 -> (a0, b0)))
      arising from a use of ‘print’
    Possible fix:
      add an instance declaration for (Show (a0 -> b0 -> (a0, b0)))
    In a stmt of an interactive GHCi command: print it
```

The error says: the type of `(,)`, which is `a0 -> b0 -> (a0, b0)` (a function[19]) is not a member of the `Show` typeclass (which is no surprise seeing we can't print functions).

So what does `(,)` do? It's safe to say that it creates a tuple from its two parameters[20]. By the same logic we have (,,), (,,,) etc.

```
ghci> (,) 5 6
(5,6)
ghci> (,) 123 "abc"
(123,"abc")
ghci> (,,) 'a' 16 "ddx"
('a',16,"ddx")
```

It's more readable to just do it normally, like `(5, 6)`. Like all prefix functions, `(,)` comes in handy for *Problem Z*.

Another thought experiment — let's imagine that somebody told us about two useful functions: `fst` and `snd`, but they didn't mention what they do. As always, we want to check their types first.

[18]We can write functions to add an element to a tuple of a specific size (and type) but never "universal" ones that work on all of them.
[19]One that takes two types and returns a tuple which contains those types.
[20]2-tuples (those made using `(,)`) are usually called pairs (or sometimes doubles), 3-tuples are triple(t)s etc.

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

Now it's clear. `fst` must take the first element of a pair, and `snd`, the second.

```
ghci> fst (5, "a")
5
ghci> snd (5, "a")
"a"
ghci> fst (1, 2, 3) -- whoops, error
```

***Warning!*** `fst` and `snd` only work on pairs. There are no built-in functions for triples or larger.

### 3.3.4. Applications

Tuples are especially useful in conjunction with functions or list comprehensions, namely when we want to return multiple things. We now go back to some of the 2.3.2 examples, and try to improve them.

```
ghci> [ (a, b) | a <- [1..3], b <- [1..3] ]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
ghci> [ (x, y, x + y) | x <- [1..4], y <- [1..3], even (x + y) ]
[(1,1,2),(1,3,4),(2,2,4),(3,1,4),(3,3,6),(4,2,6)]
ghci> take 5 [ (a, b, c) | a <- [1..], b <- [1..a], c <- [1..b], a^2 == b^2
    + c^2 ]
[(5,4,3),(10,8,6),(13,12,5),(15,12,9),(17,15,8)]
```

So far, so good. Tuples seem to be okay for trivial uses, but where they really work wonders is in larger, more complex programs. A classic example is splitting a list in order to work on both parts simultaneously. We'll look deeper into this in [XREF] and [XREF].

```
ghci> let splitHead xs = (head xs, tail xs)
ghci> splitHead [1, 5, 3, 2, 6]
(1,[5,3,2,6])
ghci> splitHead []
(*** Exception: Prelude.head: empty list
```

Of course, we can't perform `splitHead` on an empty list, because it has no head. A better, built-in function called `splitAt` solves our problems gracefully.

```
ghci> :t splitAt
splitAt :: Int -> [a] -> ([a], [a])
```

It seems that `splitAt` also takes an `Int` apart from the list, and returns a pair of lists so it's logical to think that:

1. It will split the list at any point, and
2. It won't give us unexpected errors for out-of-bounds values.

```
ghci> splitAt 5 [1..10]
([1,2,3,4,5],[6,7,8,9,10])
ghci> splitAt 1 [2, 3, 5, 8]
([2],[3,5,8])
```

```
ghci> splitAt 0 [2, 3, 5, 8]
([],[2,3,5,8])
ghci> splitAt (-1) [2, 3, 5, 8]
([],[2,3,5,8])
ghci> splitAt 5 [1, 2]
([1,2],[])
ghci> splitAt 1 []
([],[])
```

That's it for now! We'll return to types later on, but our next big step is mastering functions with advanced syntax and everything.

# Part II.

# Getting the Hang of It

# 4. Exploring Syntax

Uninformed people believe that syntax is the hardest part of learning a language.

*(kmc)*

## 4.1. Pattern Matching

### 4.1.1. Basics

We've seen the if-else in action (2.1.3). A serious downside is that it uses so much space. What if we want to create a mini-dictionary?[1]

```haskell
-- File: useless-dict.hs

engGer :: [Char] -> [Char]
engGer word = if word == "one"   then "eins"
         else if word == "two"   then "zwei"
         else if word == "three" then "drei"
         else if word == "four"  then "vier"
         else if word == "five"  then "fünf"
         else if word == "six"   then "sechs"
         else "I don't know what " ++ word ++ " means."
```

That works perfectly, apart from the fact that it looks awful and contains lots of superfluous information, such as the first `if` or the second `if` or the third `if`...

Fortunately, we can do this instead:

```haskell
-- File: patterns.hs

engGer :: [Char] -> [Char]
engGer "one"   = "eins"
engGer "two"   = "zwei"
engGer "three" = "drei"
engGer "four"  = "vier"
engGer "five"  = "fünf"
engGer "six"   = "sechs"
engGer word    = "I don't know what " ++ word ++ " means."
```

A few things to note:

- It looks much better[2].
- We don't need to align the =s but it increases readability.
- We have one function body for each use case.

[1]Bear with us — the first examples are really boring.
[2]...but it's still inefficient to write a dictionary like that.

In the second example we have used something called pattern matching. Essentially, Haskell looks at each of the patterns (from top to bottom)[3], and if one works, it will evaluate the corresponding function body. It's pretty simple if we think about it. To clarify, the syntax looks like:

```
-- Syntax: pattern matching
function pattern1 = result1
function pattern2 = result2
function pattern3 = result3
function pattern4 = result4
...
```

If we're not careful, our pattern matching can fail. This happens mostly when we don't cover all our angles — we forget to consider a case.

```
-- File: patterns-wrong.hs

intToString :: Int -> [Char]
intToString 1 = "one"
intToString 2 = "two"
intToString 3 = "three"
```

This example is boring, but it illustrates the issue quite well. It's obvious that all cases except `1`, `2`, and `3` are missing, but in real life things may not be so straightforward. GHCi throws an error when it can't find a corresponding pattern to match the input.

These errors are particularly dangerous because the compiler can't find them right away: it has to be given an incorrect input, and by that time it might be too late. We *can* use `:set -fwarn-incomplete-patterns` and GHCi will warn us on non-exhaustive patterns, but this isn't 100% guaranteed — better to check personally.

```
ghci> intToString 3
"three"
ghci> intToString 20
*** Exception: dontbother.hs:(4,1)-(6,23): Non-exhaustive patterns in
   function Main.intToString
```

***Warning!*** Make sure all possible cases are covered in pattern-matching.

The obvious solution is to introduce some sort of "catch-all" pattern.

```
-- File: patterns-wrong.hs (FIXED)

intToString :: Int -> [Char]
intToString 1 = "one"
intToString 2 = "two"
intToString 3 = "three"
intToString n = "I␣don't␣know␣about␣" ++ show n
```

In this case, everything is well. The program won't crash when we give an unexpected input, but it won't do anything useful either. As we progress, we'll learn how to deal with increasingly complex scenarios.

```
ghci> intToString 20
"I␣don't␣know␣about␣20"
```

For the avid reader, B.2.3 shows a basic method of customizing error messages — useful when we don't really want to "fix" them.

[3] If we move `engGer word = ...` at the top, it will always say "`I don't know ...`", because `word` fits anything (it's just a variable name), and is checked first.

### 4.1.2. Applications

We don't actually want to use pattern matching just as a glorified if-else. Where it really shines is in matching *patterns*, not boring numbers (although it can certainly do that as well).

Earlier (3.3.3), we wanted to do `fst` on a triple. We can't do that, but at this point we know very well that we can make our own function. Let's do it.

```haskell
-- File: patterns2.hs
fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x
```

```
ghci> fst3 ("Mike", "Adams", 23)
"Mike"
```

Now that we know it works, it's a breeze to implement the whole lot.

```haskell
-- File: patterns2.hs (CONTINUED)
snd3 :: (a, b, c) -> b
snd3 (_, y, _) = y

trd3 :: (a, b, c) -> c
trd3 (_, _, z) = z
```

Let's say we're *mathematicians* with Haskell knowledge. We have a simple task ahead of us: multiplying two 2D vectors. What does that mean? Basically we are given two pairs $(a, b)$ and $(c, d)$ — the result of the multiplication is $(a \cdot c, b \cdot d)$. Easy as pie[4]. Before learning pattern matching, we might have done something like:

```haskell
-- File: vectors.hs
mulVct :: Num a => (a, a) -> (a, a) -> (a, a)
mulVct a b = (fst a * fst b, snd a * snd b)
```

It works perfectly well (we can try it), but it's not quite what we wanted. Let's arm ourselves with patterns and try again.

```haskell
-- File: vectors.hs (FIXED)
mulVct :: Num a => (a, a) -> (a, a) -> (a, a)
mulVct (a, b) (c, d) = (a * c, b * d)
```

The end result is equivalent in both cases. The obvious difference is in readability. Even though the computer doesn't care, our human readers will be thankful of our design choices.

```
ghci> mulVct (1,2) (3,4)
(3,8)
ghci> mulVct (0,1) (5,10)
(0,10)
```

A word of warning: `Num a => (a, a) -> (a, a) -> (a, a)` is not the most general type definition out there. Because we only multiply `a` with `c` and `b` with `d`, `a` and `c` can have different types from `b` and `d`. However, in this case it doesn't make much sense — vectors should be homogenous. So, even though the compiler doesn't care, we do. So here we go:

***Warning!*** Use the most general type definition *that actually makes sense.*

Another thing: Even though, at first, they might seem like a good idea, lists aren't suitable as vectors because they have variable length.

[4]Or at least we hope so.

### 4.1.3. Matching with Cons

It is time to discover the full power of the cons operator (`:`). We've seen how `[1, 2, 3]` is the same as `1:2:3:[]` and `1:[2, 3]`. All of them are patterns that can be matched.

```haskell
-- File: cons-patterns.hs
match1 :: (Num a) => [a] -> String
match1 [x, y, z] = "List of 3 numbers with sum " ++ show (x + y + z)
match1 _ = "Nope."

match2 :: (Num a) => [a] -> String
match2 (x:y:z:[]) = "List of 3 numbers with sum " ++ show (x + y + z)
match2 _ = "Nope."

match3 :: (Num a) => [a] -> String
match3 (x:[y, z]) = "List of 3 numbers with sum " ++ show (x + y + z)
match3 _ = "Nope."
```

We will say this only once: patterns made of multiple bits must be surrounded by parentheses. `(x:y:[])` is necessary, while `([x, y])` is not.

All three functions above do the exact same thing. Although this may be interesting, in our case, their main disadvantage is that they match only lists of length 3. It's not particularly useful, but what it illustrates is the equivalence of certain notations.

Before continuing, we must note that pattern matching *cannot* be done with arbitrary functions. For example, trying it with `++` gives a parse error.

```haskell
-- File: cons-patterns-wrong.hs
match4 :: (Num a) => [a] -> String
match4 ([x,y] ++ [z]) = "List of 3 numbers with sum " ++ show (x + y + z)
match4 _ = "Nope."
```

Although it certainly looks logical to us, the compiler doesn't think the same.

```
ghci> :l cons-patterns-wrong.hs
[1 of 1] Compiling Main             ( cons-patterns-wrong.hs, interpreted )

cons-patterns-wrong.hs:3:9: Parse error in pattern: [x, y] ++ [z]
Failed, modules loaded: none.
```

The reason it works with `:` and not with `++` is that `:` creates (*cons*tructs) the list from elements, while `++` is just a function that happens to operate on lists.

We've seen how to create pattens that exactly match the input (`engGer "one"`). We've also learned that we can use variables (`intToString n`). We know that we can combine the two (`snd3 (_, y, _)`). Now we want to be able to match lists of arbitrary length[5].

We can't bind all of the elements, individually, to variables because we don't know how many of them there are. What we can do is, say, name the first element of the list, say, `x` and the rest of the elements `xs`.

```haskell
-- File: cons-patterns.hs (CONTINUED)
describe :: (Show a) => [a] -> String
describe (x:xs) = "A list with the first element " ++ show x ++ " and " ++
    show (length xs) ++ " other elements."
```

[5] After all, if we can't do that, lists are basically useless.

This works because something like `[1, 2, 3, 4, 5]` is exactly the same as `1:[2, 3, 4, 5]` so it fits the pattern `x:xs` — `x` is `1` and `xs` is `[2, 3, 4, 5]`.

```
ghci> describe [1..5]
"A list with the first element 1 and 4 other elements."
ghci> describe "hello, world"
"A list with the first element 'h' and 11 other elements."
ghci> describe []
*** Exception: cons-patterns.hs:3:1-113: Non-exhaustive patterns in
   function describe
```

What seems to be the problem? If we look closely, `[]` doesn't actually fit the pattern `x:xs`. There is no first element of `[]`, so `x` can't be matched to it. Thus the whole pattern fails (half wrong is all wrong). We can solve this right away.

```haskell
-- File: cons-patterns.hs (CONTINUED) (FIXED)
describe :: (Show a) => [a] -> String
describe [] = "An empty list."
describe (x:xs) = "A list with the first element " ++ show x ++ " and " ++
   show (length xs) ++ " other elements."
```

```
ghci> describe []
"An empty list."
```

Incidentally, the `head` function in `Prelude` is defined similarly. We can make our own!

```haskell
-- File: ourhead.hs
head' :: [a] -> a
head' (x:_) = x
head' []    = undefined
```

This `undefined` is exactly what it says on the tin: the `head'` of an empty list doesn't make sense, or, in other words, it's `undefined`.

```
ghci> head' [4, 4]
4
ghci> head' []
*** Exception: Prelude.undefined
```

Just a quick reminder: if we want to have custom error messages, we can take a look at `error`, explained in B.2.3.

### 4.1.4. "As" patterns

Observe a simple function. Its disadvantage is that we write `x:xs` twice. The interpreter essentially splits the string into a head and a tail and then puts it back together again. It's inefficient.

```haskell
-- File: as-patterns.hs
f :: String -> String -- String is the same as [Char]
f "" = "This is an empty string."
f (x:xs) = "The string " ++ x:xs ++ " has the first character " ++ [x]
```

Notice the difference (below) when using "as" patterns — by writing `all@(x:xs)` instead of simply `(x:xs)` we can reference the whole pattern by using the name `all`, without having to write `x:xs` again. This saves us from unnecessary keystrokes and the interpreter from unnecessary operations.

```
-- File: as-patterns.hs (FIXED)
f :: String -> String -- String is the same as [Char]
f "" = "This␣is␣an␣empty␣string."
f all@(x:xs) = "The␣string␣" ++ all ++ "␣has␣the␣first␣character␣" ++ [x]
```

Another example:

```
-- File: as-patterns2.hs
split3 :: [a] -> (a, a, [a])
split3 (x:y:ys) = (x, y, x:y:ys)
split3 _ = undefined
```

Last chance to learn `error` (B.2.3) — we won't be using `undefined` any longer, except in quick and dirty examples.

```
-- File: as-patterns2.hs (FIXED)
split3 :: [a] -> (a, a, [a])
split3 list@(x:y:ys) = (x, y, list)
split3 _ = error "split3:␣list␣too␣short"
```

As[6] we've stated above, writing stuff like `name@horriblyLongPattern` will bind the entire pattern to `name`, so we won't have to repeat ourselves. In this case, `list@(x:y:ys)` spares us the need to write `x:y:ys` again. We just say `list`.

### 4.1.5. Patterns in Comprehensions

Oh, just so we don't forget: we can use pattern matching in list comprehensions, too.

```
ghci> let stuff = [(4, 5), (8, 3), (2, 2), (6, 1), (3, 2)]
ghci> [ a * b | (a, b) <- stuff ]
[20,24,4,6,6]
ghci> [ a + b | (a, b) <- stuff, even a, odd b ]
[9,11,7]
ghci> [ [a, b] | (a, b) <- stuff ]
[[4,5],[8,3],[2,2],[6,1],[3,2]]
```

This time, if a pattern fails, it will just move on to the next element.

```
ghci> let newstuff = [[4,5,6], [7,8], [9,10,11]]
ghci> [ a + b*c | [a,b,c] <- newstuff ]
[34,119]
ghci> [ 2*a | [a] <- newstuff ]
[]
```

If a pattern's *type* fails, however, the result is not as pretty.

```
ghci> [ x + y | (x, y) <- [(1, 1, 1), (2, 2, 2)] ]

<interactive>:1:11:
    Couldn't match expected type ‘(t0, t1, t2)’
                with actual type ‘(t3, t4)’
    In the pattern: (x, y)
    In a stmt of a list comprehension: (x, y) <- [(1, 1, 1), (2, 2, 2)]
    In the expression: [x + y | (x, y) <- [(1, 1, 1), (2, 2, 2)]]
```

[6]Haha.

***Warning!*** While failing patterns can be excused, using the wrong type *always* results in an error.

## 4.2. Other Constructs and Expressions

### 4.2.1. Guards

We were very vehement about the fact that pattern matching is *not* a glorified if-else. The following is:

```haskell
-- File: guards.hs
numberSize :: (Ord a, Fractional a) => a -> String
numberSize x
    | x < 0.1    = "Small"
    | x < 1      = "Small-ish"
    | x < 10     = "Okay"
    | x < 100    = "Large"
    | otherwise = "Huge!"
```

In the above example, we tried to estimate the size of a given number using adjectives like `Small-ish` and `Huge!`. This is not terribly mature, but shows how these things (which, by the way, are called *guards*) look like.

Guards are basically a replacement of if-else trees. They are separated by `|`[7] and usually neatly aligned on separate lines for readability. They consist of a boolean expression (such as `x < 10`), followed by `=`, and then the result (“`Okay`”).

Just like patterns, guards are checked from top to bottom. The first boolean to be `True` has its result evaluated (and Haskell *won't* continue with the other patterns). The final guard, `otherwise`[8], is the same as writing `True`, but it looks more similar to written English, so it's preferred.

After this huge block of text, we should refresh our eyes by looking at some code. We've implemented our own versions of `max`, `min`, `abs`[9], and `compare` in a variety of styles.

```haskell
-- File: guards.hs (CONTINUED)
max2 :: Ord a => a -> a -> a
max2 x y
    | x <= y    = y
    | otherwise = x

min2 :: Ord a => a -> a -> a
min2 x y | x <= y = x | otherwise = y

abs2 :: (Num a, Ord a) => a -> a
abs2 x | x < 0     = -x
       | otherwise = x

abs2' :: (Num a, Ord a) => a -> a
abs2' x | x < 0 = -x
abs2' x         = x

compare2 :: Ord a => a -> a -> Ordering
x `compare2` y | x == y    = EQ
```

[7]These things are called pipes. We've seen them in list comprehensions but here they do entirely different things.

[8]It's not mandatory but highly recommended. If Haskell reaches the end of the guards without meeting an `otherwise`, it checks the next pattern (as in pattern matching). If no corresponding patterns are found, an error is thrown.

[9]A little more restrictive than the official implementation (requires `Ord`).

```
               | x <= y     = LT
               | otherwise = GT

compare2' :: Ord a => a -> a -> Ordering
compare2' x y       | x == y = EQ
 | x <= y = LT
           | otherwise = GT
```

All of the above are valid, but some are more readable than others. From top to bottom:

1. `max2` has a pretty standard style — we've seen this one above, and it's very readable.
2. `min2` is at the other end of the spectrum: putting guards in a single line is not a good idea.
3. `abs2` puts the guards immediately to the right of the function and starts them on the same line. Also OK.
4. `abs2'` uses a combination of guards and pattern matching. It does the same thing as `abs'`, but uses a totally different layout. Not usually recommended, but in some cases it looks better than the alternatives.
5. `compare2` is like `abs2`. What's different is that it's declared infix (surrounded by backquotes) to increase readability.
6. `compare2'`: this is very bad. It works just fine, but it looks horrendous. We also notice that the guards must be indented at least one character[10] (for the record, the recommended amount is four).

At the end of the day, it's not a big deal which style we choose[11]. It's important to be as consistent as possible, but not if it means sacrificing readability.

Let's try some more examples with guards. Say we want to make a "drink" calculator. It shows us how sober somebody is, given the blood alcohol concentration[12].

```
-- File: drink-calc.hs
drink :: (Ord a, Fractional a) => a -> String
drink bac -- Blood Alcohol Concentration
    | bac < 0.03 = "You're␣as␣sober␣as␣can␣be␣expected."
    | bac < 0.08 = "You␣can␣drive,␣but␣it's␣a␣bad␣idea."
    | bac < 0.10 = "Your␣reasoning␣is␣out␣the␣window."
    | otherwise  = "Stop␣drinking."
```

This is kinda lengthy, and not very useful, but we'll perfect it as we move along. For now, let's give it a try.

```
ghci> drink 0.07
"You␣can␣drive,␣but␣it's␣a␣bad␣idea."
ghci> drink (4/30)
"Stop␣drinking."
ghci> import Data.Ratio -- let's try rationals, too
ghci> drink (1 % 5)
"Stop␣drinking."
```

One does not simply know the blood alcohol concentration — it needs to be calculated. Fortunately, there is a simple formula, where $N$ is the number of drinks.[13]

[10] If the | starts at the very beginning of the line, Haskell treats it as a new function definition.

[11] Except the everything-on-a-single-line method (`min2`) and the one randomly indented (`compare2'`) — we run from them like the plague.

[12] We've found this information on the internet, so it's not the most precise calculator out there.

[13] *Warning! Excessive alcohol consumption can be hazardous to your health. Driving vehicles or operating heavy machinery should not be done under the influence of this dangerous chemical. Drink responsibly. Drive safely. This message brought to you by Haskellers Anonymous.*

$$c = \begin{cases} 0.025 \cdot N & \text{if you're male} \\ 0.035 \cdot N & \text{if you're female} \end{cases}$$

In Haskell speak, this is `bac = n * if sex == "male" then 0.025 else 0.035`. Apart from doing what we want it to do, this is yet another reminder that we can jam the if-else anywhere.

It's better than saying `bac = if sex == "male" then n*0.025 else n*0.035` because we're not repeating ourselves, not to mention that it's clearer.

With our current knowledge of Haskell, there are two ways of doing it, neither particularly good.

```haskell
-- File: drink-calc.hs
drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n -- Blood Alcohol Concentration
    | (n * if sex == "male" then 0.025 else 0.035) < 0.03 = "You're as sober as can be expected."
    | (n * if sex == "male" then 0.025 else 0.035) < 0.08 = "You can drive, but it's a bad idea."
    | (n * if sex == "male" then 0.025 else 0.035) < 0.10 = "Your reasoning is out the window."
    | otherwise  = "Stop drinking."
```

If we try it out, it works:

```
ghci> drink "male" 4
"Stop drinking."
ghci> drink "female" 2
"You can drive, but it's a bad idea."
ghci> drink "male" 1
"You're as sober as can be expected."
ghci> drink "female" 8
"Stop drinking."
```

The code is, however, yucky (and that's putting it mildly). The other solution is to use another function to calculate the `bac`.

```haskell
-- File: drink-calc.hs (FIXED)
bac :: (Fractional a, Ord a) => String -> a -> a
bac sex n = n * if sex == "male" then 0.025 else 0.035

drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n -- Blood Alcohol Concentration
    | bac sex n < 0.03 = "You're as sober as can be expected."
    | bac sex n < 0.08 = "You can drive, but it's a bad idea."
    | bac sex n < 0.10 = "Your reasoning is out the window."
    | otherwise  = "Stop drinking."
```

It still works and it's a tad shorter, but that's about it. We're still repeating ourselves and we've just introduced a function that we're not going to use anywhere else. With what we know so far, there's nothing we can do.

### 4.2.2. Where Bindings

This is where `where` bindings come into play. We're not going to improve `bac` right away — let's start with an example.

```
-- File: gpa.hs
gpa :: [Int] -> Int -> Int
gpa grades final = func grades + final
    where func :: [Int] -> Int
          func xs = sum xs `div` length xs
```

It is time to take a moment and contemplate this function.

Okay, moment's over. So what do we have here? Why, a GPA calculator, of course. This one seems to do something with the grades then add it to the final. If we only read the first line, we don't know what `func` does. Neither does the compiler.

The `where` keyword introduces a section that contains definitions. In our case, `func` is defined just like we learned. It's easy to see what it does. The type definition tells us that it takes a list of integers and returns only one, and the body indicates it averages those numbers[14]. So `gpa` adds the final to the average of the other grades. Pretty simple.

Another thing: inside `where` sections we can have the usual gimmicks: type declarations (which are usually omitted[15]), multiple function bodies, pattern matching etc. It's just like our typical function (or name) definition. We can even put a `where` inside a `where`!

In fact, pattern matching inside where sections is so useful and important, it's worth giving a specific example.

```
-- File: stutter.hs
stutter :: String -> String
stutter word = [w] ++ "-" ++ [w] ++ "-" ++ word
    where (w:_) = word
```

It's `[w]`, not `w` because `++` takes strings, not characters. The keen reader would notice that we can also do things like `where w = head word`. No matter how we write it, we should be consistent with our choices.

```
ghci> stutter "hello"
"h-h-hello"
```

These are the basics of `where` bindings. Now it's time to improve our calculator (in three easy steps). This is the initial code:

```
bac :: (Fractional a, Ord a) => String -> a -> a
bac sex n = n * if sex == "male" then 0.025 else 0.035

drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n
    | bac sex n < 0.03 = "You're␣as␣sober␣as␣can␣be␣expected."
    | bac sex n < 0.08 = "You␣can␣drive,␣but␣it's␣a␣bad␣idea."
    | bac sex n < 0.10 = "Your␣reasoning␣is␣out␣the␣window."
    | otherwise        = "Stop␣drinking."
```

Problems:

- We're repeating ourselves.
- We have a function that we use nowhere else.
- The code is slightly confusing.

The obvious thing to do is put `bac` in a `where` section (not to worry, the `where` is visible to all the guards).

[14] We should have called it `average` or `avg` or something instead of `func`.
[15] Because functions inside `where` sections are usually short and simple. If one becomes too long, consider writing it separately.

```haskell
drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n
    | bac sex n < 0.03 = "You're as sober as can be expected."
    | bac sex n < 0.08 = "You can drive, but it's a bad idea."
    | bac sex n < 0.10 = "Your reasoning is out the window."
    | otherwise  = "Stop drinking."
    where bac :: (Fractional a, Ord a) => String -> a -> a
          bac sex n = n * if sex == "male" then 0.025 else 0.035
```

Problems:

- We're repeating ourselves.
- ~~We have a function that we use nowhere else.~~
- The code is slightly confusing.

Now we get rid of `bac`'s type declaration — the function is simple enough. We also notice that `sex n` is redundant (`drink` already has the parameters `sex` and `n`, which can be used in the `where` section).

```haskell
drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n
    | bac < 0.03 = "You're as sober as can be expected."
    | bac < 0.08 = "You can drive, but it's a bad idea."
    | bac < 0.10 = "Your reasoning is out the window."
    | otherwise  = "Stop drinking."
    where bac = n * if sex == "male" then 0.025 else 0.035
```

Problems:

- ~~We're repeating ourselves.~~
- ~~We have a function that we use nowhere else.~~
- The code is slightly confusing.

Finally, let's make the function easier to understand and modify by giving names to `0.03`, `0.08` and `0.10`. This way we can be sure we understand what they mean and also easily modify them (for instance, France has a `0.05` limit for driving).

```haskell
drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n
    | bac < soberLimit    = "You're as sober as can be expected."
    | bac < drivingLimit  = "You can drive, but it's a bad idea."
    | bac < thinkingLimit = "Your reasoning is out the window."
    | otherwise           = "Stop drinking."
    where bac = n * if sex == "male" then 0.025 else 0.035
          soberLimit    = 0.03
          drivingLimit  = 0.08
          thinkingLimit = 0.10
```

Problems:

- ~~We're repeating ourselves.~~
- ~~We have a function that we use nowhere else.~~
- ~~The code is slightly confusing.~~

Now we're ready to move on. Oh, and one more thing. We must align things neatly following the `where` part, or the code might not compile or function correctly.

***Warning!*** In `where` sections, not aligning the code can yield undesirable results.

However, placing the `where` on a separate line is allowed, like in the following example:

```
-- File: cone.hs
coneVolume :: Floating a => a -> a -> a
coneVolume r h = baseArea * h / 3
    where
        baseArea = pi * r^2
```

### 4.2.3. Let Bindings

We'll recycle the above example for our purposes.

```
-- File: cone-let.hs
coneVolume :: Floating a => a -> a -> a
coneVolume r h =
    let baseArea = pi * r^2
    in  baseArea * h / 3
```

It seems pretty intuitive. One might say `let` bindings are just like `where` bindings, only with the order reversed — `let <bindings> in <expression>`, as opposed to `<expression> where <bindings>`. There's much more to them, though. A mountain of examples follows (and not many words).

For a start, `let` is not unlike the `if` statement; we can jam it pretty much everywhere — interactive...

```
ghci> let a = 3 in 2 * a
6
ghci> 4 + 5 * (let x = 5 in 2 * x)
54
ghci> 2 + 3 * (let e = 2.718281828 in e * (e + 1))
32.32201377330506
ghci> "hello" ++ (let w = "␣world" in w ++ w ++ w)
"hello␣world␣world␣world"
```

... and loaded from a file (just like `where` bindings, `let` bindings must be properly aligned).

```
-- File: cone-area.hs
coneArea :: Floating a => a -> a -> a
coneArea r h =
    let baseArea = pi * r^2
        sideArea = let l = sqrt (r^2 + h^2) in pi * r * l
    in  baseArea + sideArea
```

We can perform many neat tricks using `let`, such as:

- Binding several variables *inline*[16] using semicolons.

```
ghci> let x = 4; y = 5; z = 6 in (x + y) * z
54
ghci> "Hello␣" ++ (let x = "world"; y = "wide␣" in y ++ x) ++ "!"
"Hello␣wide␣world!"
```

- Using pattern matching

[16] A fancy way of saying "in (the middle of) a single line".

```
ghci> let (x, y) = (3, 2) in y * x
6
ghci> let x:y:_ = "asdf" in y:x:[]
"sa"
ghci> 4 + (let a:b:c:_ = [5,10..] in c - b + a)
14
```

- Putting them inside list comprehensions

```
ghci> [ x | x <- [1..10], let a = 8*x, a < 50]
[1,2,3,4,5,6]
ghci> [ x:xs | x <- ['a'..'c'], let xs = "ghj"]
["aghj","bghj","cghj"]
```

- Nesting them.

```
ghci> let x = 4 in let y = 5 in x + y
9
ghci> let a = 'h' in let as = "ello" in a:as
"hello"
```

When defining several variables with `let`, we can use one in the definition of another.

```
ghci> let x = 4; y = 2*x in x + y
12
ghci> let x = 5; y = 3 + x; z = x * y in x + y - z
-27
```

We can also do it in any order.

```
ghci> let y = 2*x; x = 4 in x + y
12
ghci> let y = 3 + x; z = x * y; x = 5 in x + y - z
-27
```

It won't work, however, in separate `let`s or if we try to use a variable prior to its let binding.

```
ghci> [ x | x <- [1..10], y < 2, let y = x - 5]

<interactive>:1:21: Not in scope: ‘y’
ghci> let y = 2 * x in (let x = 4 in y + x)

<interactive>:2:13: Not in scope: ‘x’
```

Additionally, `let` bindings are *not* visible across guards. All these drawbacks are the result of a very simple things: `let` bindings are very "local"; they are only visible where we define them — we talk more about local things in A.2.1. For instance:

```
Prelude> let a = 3 in 2 * a
6
Prelude> a

<interactive>:2:1: Not in scope: ‘a’
ghci> (let b = 5 in 4 * b) + b
```

```

<interactive>:3:24: Not in scope: `b'
ghci> [ x | x <- [1..10], let c = 2*x, c < 5] ++ [c]

<interactive>:4:45: Not in scope: `c'
```

There is only one exception to this rule: we can omit the `in` part when defining things interactively; this way, the names will be visible during the entire interactive session (but not the next).

```
ghci> let a = 5; b = 6
ghci> "hello world"
"hello world"
ghci> a + b
11
ghci> :q
Leaving GHCi.
ee@bt:~$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude> a + b

<interactive>:2:1: Not in scope: `a'

<interactive>:2:5: Not in scope: `b'
```

It's time for a little discussion and recap.

The `let <bindings> in <expression>` syntax allows `let` to be put anywhere, especially inside larger expressions. That's the most important difference between `let` and `where`.

Interestingly, `let` bindings are so local, that it somehow limits their usefulness[17]. Coincidentally, this is also one of their great advantages.

The above reasons, and more, bring us to our final point: `let` and `where` are not always interchangeable — `where` is better with guards; `let`, inside larger expressions.

### 4.2.4. Bonus: Case Expressions

Just like `[1, 2, 3]` is syntactic sugar for `1:2:3:[]`, pattern matching (in function definitions) is just syntactic sugar for case expressions.

```haskell
-- File: case-expr.hs
tail' :: [a] -> [a]
tail' [] = error "tail': empty list"
tail' (_:xs) = xs
```

We've just implemented our version of `tail` using pattern matching (in function definitions). Let's see how it looks with case expressions.

```haskell
-- File: case-expr.hs (FIXED)
tail' :: [a] -> [a]
tail' all = case all of [] -> error "tail': empty list"
                        (_:xs) -> xs
```

[17]The biggest problem is that they won't work with guards the way we want them to.

The syntax for case expressions is pretty much self-explanatory. A longer example, just to consolidate our knowledge:

```
-- File: case-expr2.hs
f :: Int -> String
f n = case n of 1 -> "one"
                2 -> "two"
                _ -> "many"
```

Of course, those can be any patterns, not just numbers. If it's not 100% clear yet, this is the syntax:

```
-- Syntax: case expressions (in function definitions)
function argument = case argument of pattern1 -> result1
                                     pattern2 -> result2
                                     pattern3 -> result3
                                     pattern4 -> result4
                                     ...
```

We've been very careful to mention "in function definitions" repeatedly. That's because, technically, case expressions make use of pattern matching, so it's not really fair to compare the two.

Their main advantage is that case expressions work anywhere, just like `let` bindings. Basically, they enable pattern matching anywhere we desire. We can put them in the middle of an expression, for example.

```
-- File: case-expr3.hs
f :: (Show a) => [a] -> String
f []    = "This␣list␣is␣empty.␣Sorry."
f [x]   = "This␣list␣is␣a␣singleton,␣with␣the␣element:␣" ++ show x
f (x:_) = "This␣list␣is␣longer.␣Its␣head␣is:␣" ++ show x
```

```
-- File: case-expr3.hs (FIXED)
f :: (Show a) => [a] -> String
f xs = "This␣list␣is␣" ++ case xs of []    -> "empty.␣Sorry."
                                     [x]   -> "a␣singleton,␣with␣the␣
                                        element:␣" ++ show x
                                     (x:_) -> "longer.␣Its␣head␣is:␣" ++
                                        show x
```

The reason we don't use case expressions all the time is much like the reason we don't abuse `let` bindings: they are ever-so-slightly less readable[18] than the alternatives. Syntactic sugar in general offers a clearer exposition at the expense of power.

In fact, after this chapter on syntax, we've seen many alternative ways of solving a given problem. Which one to use is left at the reader's discretion.

[18]Some people may disagree.

# 5. Recursion

Primitive recursion is the goto of functional programming.

*(anonymous)*

## 5.1. Basic Implementation

### 5.1.1. Understanding Recursion

Recursion is perhaps one of the most powerful tools in all of Haskell[1]. According to Wikipedia, recursion is the process of repeating items in a self-similar way. In programming, recursion is a method of defining functions in which the function is applied within its own definition. Simply put, a recursive function is a function that calls itself.

To understand the principle, this chapter concerns itself only with explicit (also called primitive) recursion — the easiest and most basic form of recursion. Later (in [XREF]) we will see many cool functions that perform recursion for us.

The simplest example is the factorial[2]. We can write `factorial n = product [1..n]`, but that's not the definition we're looking for. This is:

```haskell
-- File: factorial.hs
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

```
ghci> factorial 3
6
ghci> factorial 5
120
```

It works, *but why*? Let's see what GHCi does if we try to call `factorial 4`.

1. `factorial 4` is `4 * factorial 3`.
2. `factorial 3` is `3 * factorial 2`, so `factorial 4` is `4 * (3 * factorial 2)`.
3. `factorial 2` is `2 * factorial 1`, so `factorial 4` is `4 * (3 * (2 * factorial 1))`.
4. `factorial 1` is `1 * factorial 0`, so `factorial 4` is `4 * (3 * (2 * (1 * factorial 0)))`.
5. `factorial 0` is `1`, so `factorial 4` is `4 * (3 * (2 * (1 * 1)))`.
6. `factorial 4` is `4 * (3 * (2 * 1))`.
7. `factorial 4` is `4 * (3 * 2)`.
8. `factorial 4` is `4 * 6`.
9. `factorial 4` is `24`.

[1] Author's note: it took all my willpower not to start with a recursion joke[1].

[2] The factorial of a (non-negative) integer $n$ is the product of integers from 1 to $n$. The factorial of 0 is, by convention, 1.

10. Done!

At this point, it's useful to make our line-by-line analysis. Here's the function again, without that pesky first line comment:

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

1. The type definition is important; the factorial doesn't make sense over non-integers[3]. Actually, it doesn't work on negative numbers either (which we'll discuss in 5.3.1).
2. Without this line, the function would never finish. `factorial 1` would be `1 * (0 * (-1 * (-2 ...`. This is called the "base case" or "edge condition". We'll discuss it in a moment.
3. This one puts an operation on hold (namely multiplication), then brings the evaluation closer to the base case. Eventually it will reach it, the pending operations will be performed, and the computation will end, as seen in the elaboration above.

Sounds complicated? Because it is. The above operations aren't meant to be our concern. The compiler can do them without our help. We should understand recursion intuitively, and to do that, we must think simpler.

Here's a little something to break the wall of text, and then we'll move on.

```
________                                      _____
___  __ \_____ ___________  _____________________(_)______ _______
__  /_/ /_  _ \_  ___/_  / / /__  ___/__  ___/__  / _  __ \__  __ \
_  _, _/ /  __// /__  / /_/ / _  /    _(__  ) _  /  / /_/ /_  / / /
/_/ |_|  \___/ \___/  \__,_/  /_/     /____/  /_/   \____/ /_/ /_/
```

The bottom line is, a recursive function has two main elements:

1. The "base case" — the simplest one, where we already know the answer. The base case is where the calculation ends. Some examples:

   a) The factorial of 0 is 1. We know this because it's convention. Can it get any simpler? Not really.

   b) The length of an empty list is 0. We know that because it's obvious.

   c) The maximum of a single number is that number.

2. All other cases — here we must bring evaluation closer to the base case. We must simplify. Why? Because the base case is the only way our calculation can finish. We must reach it. To reach it, we must get closer. Some examples:

   a) The factorial of $n$ is $n$ times the factorial of $n-1$; $n-1$ is closer to 0, so we're on the right track.

   b) The length of a list is one plus *the length of the list without the first element*; if we repeat this enough times, we'll reach the empty list, as planned.

   c) The maximum of a list is the first element or *the maximum of the list without the first element*, whichever is larger.

In all three situations, the regular cases bring us closer to the edge condition (base case), thus guaranteeing that the computer will, in fact, finish calculating and provide a result.

[3]Actually it does — it's called the Gamma function.

### 5.1.2. Practical Examples

It is time to put the above into code. The factorial has already been done. Let's try the length[4] one.

```haskell
-- File: length.hs
length' :: [a] -> Int
length' [] = 0
-- what are we supposed to do now?
```

Obviously, the list with the first element is one longer then the list without it. We should somehow write this down, but to do it, we must separate the list into its first element and the rest. Do we know something that does that?

Yes, it's the `x:xs` pattern. We've already covered some of its uses, but here is a quick refresher:

```haskell
-- File: xxs.hs
super :: String -> String
super (x:xs) = "First␣letter:␣" ++ [x] ++ ";␣the␣rest:␣" ++ xs
```

```
ghci> super "Greetings!"
"First␣letter:␣G;␣the␣rest:␣reetings!"
```

Now we can state the obvious, clearly and concisely.

```haskell
length' (x:xs) = 1 + length' xs
```

And that's it! If we put it inside our original code, it works like a charm.

```haskell
-- File: length.hs (FIXED)
length' :: Num a => [b] -> a
length' [] = 0
length' (x:xs) = 1 + length' xs
```

```
ghci> length' [1,2,3,4]
4
ghci> length' "haskell"
7
```

We might even notice that we're not using `x` (from the `x:xs`), so we can write `length' (_:xs)`.

To determine the maximum of a list, we have to, once again, separate the list into a head and a tail. This time we get to see the completed code directly.

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

In a dramatic twist of events, this function has *two* edge conditions. The first will be reached if (and only if) we supply `[]` — the maximum of an empty list doesn't make sense. The other one is the "normal" base case we all know and love — the maximum of a single element is itself.

The third pattern compares the head with the maximum of the tail to determine which one is bigger. Notice how `max` operates on 2 elements while `maximum'` works on an entire list.

```
ghci> maximum' [1,3,4,2,5,2]
5
```

[4]We're using `length'` because `length` already exists, and we must have a different name.

### 5.1.3. More Parameters

A recursive function can take any number of parameters. Knowing that, we'll try to implement our own `replicate`. `replicate` repeats an element a specified number of times (so 2 parameters).

```
ghci> :t replicate
replicate :: Int -> a -> [a]
ghci> replicate 5 2
[2,2,2,2,2]
ghci> replicate 6 'a'
"aaaaaa"
```

It's easier if we try to implement it for a certain element, say `'A'`. Our edge condition is trying to repeat it 0 times. We'll call the function `screamer`[5].

```
-- File: screamer.hs
-- replicate when applied to the letter 'A'
screamer :: Int -> String
screamer 0 = [] -- it's the same as ""
screamer n = 'A' : screamer (n-1)
```

Obviously, `replicate` works with any element — if we pass it as an extra parameter it should work.

```
-- File: replicate.hs
replicate' :: Int -> a -> [a]
replicate' 0 _ = []
replicate' n x = x : replicate' (n-1) x
```

```
ghci> replicate' 3 'b'
"bbb"
ghci> replicate' 2 "Hi"
["Hi","Hi"]
```

This time, one of the parameters (namely, the second one) always remained unchanged. But it is not always so. We can manipulate several parameters when writing a recursive function. This very dumb implementation of `compare`, which only works on positive integers, is a... good(ish) example.

```
-- File: dumb-compare.hs
cmp :: Integer -> Integer -> Ordering
cmp 0 0 = EQ
cmp 0 _ = LT
cmp _ 0 = GT
cmp x y = cmp (x-1) (y-1)
```

This example also illustrates a good rule of thumb[6]: the number of base cases is usually equal to the number of possible outcomes. In this case, it's three: `EQ`, `LT` and `GT`.

Anyway, the principle of this function is very simple. It decrements *both* parameters, until one reaches zero. The other is larger. Is there an even more inefficient version of `compare`? I have no idea.

`take` takes taking elements from a list to a whole new level. Example, then code.

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

[5] `replicateA` might sound tempting, but it's already taken (see [XREF]).
[6] Not to be followed blindly.

```haskell
-- File: take.hs
take' 0 _  = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs
```

Notice how the two outcomes become base cases. We either

- take 0 elements from a list, or
- try to take elements from an empty list.

In both cases, the result is `[]`. The general case is very simple, too. Taking `n` elements from a list is basically taking the first element, then `n-1` elements from the rest of the list.

Next up, `zip`. This function takes two lists and combines them together into a list of pairs. It stops when one of the lists is empty, so `zip "abc" [1, 2]` is `[('a',1),('b',2)]`.

The two edge conditions correspond to empty lists (the first and the second, respectively). The general case separates both lists in a head and a tail.

```haskell
-- File: zip.hs
zip' :: [a] -> [b] -> [(a, b)]
zip' [] _ = [] -- First list empty
zip' _ [] = [] -- Second list empty
zip' (x:xs) (y:ys) = (x, y) : zip' xs ys
```

## 5.2. Variations

### 5.2.1. Using Guards

If we're not careful, we might as well end up with a function that runs indefinitely, or worse[7]. This usually happens if the edge condition is poorly written, or if the general case does not lead to the edge condition. Half the functions we've written so far have some sort of problem. That's not very encouraging.

Our version of `replicate` (also, `screamer`) weirds out when we give it a negative number of repetitions. The predefined function works fine.

```
ghci> replicate' (-2) 5
[5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,^CInterrupted.
ghci> replicate (-2) 5
[]
```

***Warning!*** Make sure your function behaves correctly even on unexpected input.

The problem? Our edge condition should also check for negative numbers. The easy way to do it is to use a guard.

```haskell
-- File: replicate.hs (FIXED)
replicate' :: Int -> a -> [a]
replicate' n _ | n <= 0 = []
replicate' n x          = x : replicate' (n-1) x
```

This is one of the few acceptable uses of inline guards. Notice the absence of an `otherwise` clause. This is because if evaluation reaches the end of the guards, it will fall down to the next pattern (which, in our case, catches everything).

In this instance, we can also use `otherwise` and a single function body.

[7] What's worse than an infinitely running program? A wrong result.

```haskell
-- File: replicate2.hs
replicate' :: Int -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x : replicate' (n-1) x
```

Sometimes the function does something unimaginable. Our stupid `cmp` is flat-out wrong on negative numbers. The relevant parts, then illustration:

```haskell
cmp 0 0 = EQ
cmp 0 _ = LT
cmp _ 0 = GT
cmp x y = cmp (x-1) (y-1)
```

```
ghci> cmp 2 3
LT
ghci> cmp (-2) 3
GT
ghci> cmp (-2) (-3)
^CInterrupted.
```

Why does this happen? The program assumes that the first number to reach 0 is smaller. But if we decrease an already negative number, it will never become 0. So the other one will be 0 first, and will be declared the smallest. If both are negative, then the function will continue to run, and run, and run (until we run out of memory)[8].

Here is the corrected function:

```haskell
-- File: dumb-compare.hs (FIXED)
cmp :: Ord a => a -> a -> Ordering
cmp x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

The dumb implementation is doomed. There is no way we can get something usable out of it, so we should just trash it.

### 5.2.2. Multiple "Regular" Cases

Some recursive functions have different behavior for different types of input, say, even and odd numbers. This means that we have several separate cases. This can be easily achieved by using pattern matching or guards.

The classic example is the Collatz sequence. Take a positive integer.

- If it's even, divide it by two.
- If it's odd, multiply it by three and add one.

It is thought (but not proven) that after a finite number of steps, all numbers will eventually reach 1. By virtue of this fact, we know our edge condition. The two regular cases are for even and odd, respectively.

[8]Experienced programmers out there: `Integer` is unbounded, so it will never wrap around.

```
-- File: collatz.hs
collatz :: Integral a => a -> [a]
collatz 1 = [1]
collatz n
    | even n     = n : collatz (n `div` 2)
    | otherwise = n : collatz (3*n + 1)
```

This function is especially dangerous because we don't actually know if it will finish. Still, let's take it for a spin.

```
ghci> collatz 5
[5,16,8,4,2,1]
ghci> collatz 20
[20,10,5,16,8,4,2,1]
```

Of course, we can simply check the lengths. Some inputs are especially pesky[9].

```
ghci> length (collatz 27)
112
ghci> length (collatz 6171)
262
```

### 5.2.3. Infinite Recursion

It's easier than it looks. Haskell already supports infinite lists, so it should be a breeze to write versions of the following two functions:

- `repeat` repeats an element an infinite number of times
- `cycle` repeats an entire list

The easy way to do it is to simply omit the edge condition, like this:

```
-- File: inf-recursion.hs
repeat' :: a -> [a]
repeat' x = x : repeat' x

cycle' :: [a] -> [a]
cycle' xs = xs ++ cycle' xs
```

Without a base case, the function is all but guaranteed to run indefinitely. That is, unless we `take` a finite number of elements (because of laziness).

```
ghci> take 5 (repeat' 0)
[0,0,0,0,0]
ghci> take 10 (cycle' [1, 2, 3])
[1,2,3,1,2,3,1,2,3,1]
```

## 5.3. Further Expansion

### 5.3.1. Using Natural Numbers [FIXME-move to adv. types]

Every time we used some sort of "counter" which we decreased until it reached zero, we used some sort of integer. Recall the factorial function:

[9]The Online Encyclopedia of Integer Sequences has collected a list specially for the purpose: A006877.

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

But, as we mentioned, the factorial doesn't make much sense over negative numbers. In 5.2.1 we even pointed out that such functions might even run indefinitely on negatives. In that spirit, the solution is:

```haskell
factorial :: Integral a => a -> a
factorial n | n < 0 = error "factorial over negative numbers"
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

That's more of a workaround rather than a fix, however. Someone casually looking at the type definition might imagine that the function works over all integers. This is obviously not the case.

The "right" way to do it is to use the appropriate type for the function; something like `Nat` representing natural numbers would be welcome. This is a hypothetical example; no such type exists in the standard libraries.

[FIXME]

```haskell
factorial :: Nat -> Nat
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

### 5.3.2. Application: Quicksort

We have tried to postpone this moment as long as possible. It's time for the most overused piece of Haskell code in history: `quicksort`.

- What it does: it sorts a list (duh).
- How it does it: a sorted list is[10] the list with
  - the elements less than or equal to the head, *sorted*, followed by
  - the head of the list, followed by
  - the elements greater than the head, *sorted*.
- What's interesting for us is that we must call `quicksort` twice in its definition (once for the smaller elements and once for the larger ones)

So, without further ado:

```haskell
-- File: quicksort.hs
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = lesserSorted ++ [x] ++ greaterSorted
    where lesserSorted  = quicksort [ y | y <- xs, y <= x ]
          greaterSorted = quicksort [ y | y <- xs, y >  x ]
```

```
ghci> quicksort [4,1,5,3,8,7]
[1,3,4,5,7,8]
ghci> quicksort "the five boxing wizards jump quickly"
"     abcdeefghiiiijklmnopqrstuuvwxyz"
```

[10]We can't say "it does this, then it does that", because it defeats the purpose of functional programming, which emphasizes how things are defined, rather then how they are done.

This implementation of quicksort is surprisingly easy to understand. The function will take the head of the list, `4` and then put it between `[1,3]` and `[5,8,7]` (after they've been sorted).

Such an algorithm is called "divide and conquer" because it literally[11] breaks the input into two easier-to-manage halves, each of them broken down even more, until we reach empty lists, which are already sorted. The pieces are then put back together in the correct order.

Unfortunately, if we perform the detailed breakdown on this function, we clearly see that the algorithm performs many useless operations (concatenating all those empty lists), so it might not be terribly efficient. [FIXME-double check]

```
-- Evaluation steps
quicksort [4,1,5,3,8,7] = quicksort [1,3] ++ [4] ++ quicksort [5,8,7]
    quicksort [1,3] = quicksort [] ++ [1] ++ quicksort [3]
        quicksort [] = []
        quicksort [3] = quicksort [] ++ [3] ++ quicksort []
            quicksort [] = []
            quicksort [] = []
    quicksort [5,8,7] = quicksort [] ++ [5] ++ quicksort [8,7]
        quicksort [] = []
        quicksort [8,7] = quicksort [7] ++ [8] ++ quicksort []
            quicksort [7] = quicksort [] ++ [7] ++ quicksort []
                quicksort [] = []
                quicksort [] = []
            quicksort [] = []
[] ++ [1] ++ [] ++ [3] ++ [] ++ [4] ++ [] ++ [5] ++ [] ++ [7] ++ [] ++ [8]
   ++ []
[1,3,4,5,7,8]
```

Indeed, running `quicksort` on `[100000,99999..1]` takes quite some time and maxes out the memory. From now on, we'll just import `Data.List`, which conveniently contains an efficient sorting function, `sort`[12]. For more `Data.List` goodies, see C.1.

```
ghci> import Data.List
ghci> sort [3,5,8,2,1]
[1,2,3,5,8]
```

### 5.3.3. Discussion

All of the functions that we have implemented in this chapter have some common ground. For instance:

- Separating a list into a head and a tail until we reach `[]`.
- Having some number and then decreasing it until it becomes 0.
- Breaking down a list into several smaller parts.

By far the most widely used data structure in this chapter was the list. Somehow lists lend themselves to being recursed upon simply because of the convenient `x:xs` pattern which, on one hand, extracts an element which can be used and, on the other hand, leaves the rest of the list available for further operations.

One of the main development directions in Haskell is abstraction. Sadly, in this book, this path has been so far left unexplored (because we were busy understanding syntax). Specifically, the primitive (explicit) recursion we have performed so far in this chapter allows us to consider only particular cases. For instance, this is an implementation of `sum`:

[11] Figuratively.
[12] Based on mergesort.

```
sum []     = 0
sum (x:xs) = x + sum xs
```

And this is an implementation of `product`:

```
product []     = 1
product (x:xs) = x * product xs
```

The function `and` operates on booleans, and tells us if all of them are `True`. Here it is:

```
and []     = True
and (x:xs) = x && and xs
```

Likewise, the function `or`:

```
or []     = False
or (x:xs) = x || or xs
```

A pattern emerges. All these concrete examples have the same basic structure, *but we do not yet know how to take advantage of it.* There must be a function that covers all these use cases. There is.

We've barely scratched the surface.

# 6. Advanced Functions

I've come to see the power of Haskell at last.
You have to treat functions like crap.

*(nikki93)*

## 6.1. Currying and Partial Application

### 6.1.1. Fundamentals

Every function in Haskell takes exactly one parameter. Multiple-parameter functions exist because of what is officially called *currying* — it's very clever. Let's refer to our first *Problem Z* example (way back, in 1.2.4).

```
compare 2 3   -- works
compare (2 3) -- doesn't work
(compare 2) 3 -- works!!
```

We've learned why the first one works and the second doesn't: spaces are used for function application and parentheses for grouping, not the other way around.

To see why the third one works, we must understand what `compare 2 3` does. It first takes the parameter 2 and returns *a function* that takes a parameter and compares 2 with it. *That function* is then applied to 3 and it finally returns `LT`. Read that again.

If we take a look at `compare`'s type, it's `compare :: Ord a => a -> a -> Ordering`. Up until now, we've said that it takes two parameters.

But now we realize that `a -> a -> Ordering` is the same as `a -> (a -> Ordering)`. So the function, in fact, takes only one parameter (an `a`) and returns an `a -> Ordering`, which is a function (that takes an `a` and returns an `Ordering`).

Let's discuss a clearer example.

```
-- File: currying.hs
addFour :: Int -> Int -> Int -> Int -> Int
-- we can also write Int -> (Int -> (Int -> (Int -> Int)))
addFour x y z t = x + y + z + t
```

Now if we add parameters one at a time:

```
ghci> :t addFour
addFour :: Int -> Int -> Int -> Int -> Int
ghci> :t addFour 1
addFour 1 :: Int -> Int -> Int -> Int
ghci> :t addFour 1 2
addFour 1 2 :: Int -> Int -> Int
ghci> :t addFour 1 2 3
addFour 1 2 3 :: Int -> Int
ghci> :t addFour 1 2 3 4
addFour 1 2 3 4 :: Int
```

Every time we add another parameter, the type gets "eaten up" from the left. That is because if we call a function with too few parameters, we'll get a function that takes the rest of them. This is called *partial application*. In other words, if `f` "takes" $n$ parameters[1]: `a1`, `a2`, `a3`, ..., `an`, then:

- `f a1` "takes" $n-1$ parameters: `a2`, `a3`, `a4`, ..., `an`
- `f a1 a2` "takes" $n-2$ parameters: `a3`, `a4`, ..., `an`
- etc.

This is also the chief reason why everything is separated by `->` in type declarations. If we clearly distinguished the parameters from the return type, we couldn't have parially applied functions and thus, indirectly, we wouldn't be able to do other neat things, like name them.

```
ghci> let compare2With = compare 2
ghci> compare2With 5
LT
ghci> compare2With 1
GT
ghci> compare2With 2
EQ
```

Do we know some other way of defining `compare2With`? Of course, `compare2With x = compare 2 x`. We've done things this way many times before. I know we're repeating ourselves, but let's see them again.

```
compare2With x = compare 2 x -- the way we've done things
compare2With   = compare 2   -- equivalent to the above
```

Notice how `x` was present on the *right* side on both hand-sides of the first equation. Therefore, `x` is superfluous (it can safely be removed). Watch out, though, because in something like `compare2With x = compare x 2` (`x` on the left), `x` can't be eliminated without changing the meaning.

***Warning!*** Partial application only occurs from left to right (beginning with the first parameter).

Actually it's pretty difficult to explain rigorously. It's something that is very intuitive but nevertheless hard to elaborate. It's like in mathematics. We can say that "the function $f$ applied to $x$ adds 2 to $x$" or we can simply state "the function $f$ adds 2". It's implied that it adds 2 to its parameter[2].

So there you have it. Currying is often confused with partial application, but they are really quite different:

- Currying is what makes a function take only one parameter and return a function that takes another parameter and so on. We'll discuss it a little later, in [XREF].
- Partial application is the act of supplying a function with too few arguments.

Currying and partial application are two of the most important concepts in all of Haskell, so it's a good idea to be familiar with them.

### 6.1.2. Problem Z

We've put all the cool things that happen because of currying and partial application under the umbrella term *Problem Z*. Now it's time to revisit them.

In 2.1.3 we said that a constant really is a zero-parameter function. It makes sense if we think about it — there are no parameters for us to change so the result will always be the same. Do we know what else takes zero parameters? A fully-applied function. Take `compare 2 3` for instance.

[1]We're going to say that a function takes $n$ parameters for simplicity, even though we know what's actually going on.
[2]What else can it add 2 to?

```
ghci> :t compare 2 3
compare 2 3 :: Ordering
ghci> :t LT
LT :: Ordering
ghci> LT == compare 2 3
True
```

Moving on, when we discussed infix functions (in 2.1.4) we illustrated how infix functions can be called prefix.

```
ghci> 2 + 3
5
ghci> (+) 2 3
5
```

This enables us to partially apply them[3].

```
ghci> :t (+) 2
(+) 2 :: Num a => a -> a
```

However, there is a simpler, more intuitive way, by using *sections*. Simply put, we omit one of the sides:

```
ghci> :t (2/)
(2/) :: Fractional a => a -> a
ghci> :t (/2)
(/2) :: Fractional a => a -> a
```

We still have to put them in parentheses because otherwise the compiler will treat them as incomplete expressions.

Sections have another advantage. Notice the difference between the following two:

```
ghci> (2/) 3
0.6666666666666666
ghci> (/2) 3
1.5
```

In the second example, we've partially applied the *second* parameter. Neat, huh?

Speaking of sections, we might be tempted to do something like `(3,) 2`, but the compiler will scream at us.

```
ghci> (3,) 2

<interactive>:1:1: Illegal tuple section: use -XTupleSections
```

What GHCi means by this is that it recognizes what we're trying to do, but won't allow it. It also mentions that if we open GHCi with the option `-XTupleSections`, it will work just fine.

```
ee@bt:~$ ghci -XTupleSections
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude> :set prompt "ghci>␣"
ghci> (3,) 2
(3,2)
```

But why bother when we can just use `(,)` instead?

[3]This is not the main advantage, however. Details in [XREF].

```
ghci> :t (,) 3
(,) 3 :: Num a => b -> (a, b)
ghci> (,) 3 2
(3,2)
```

### 6.1.3. When It's Not

[FIXME-need to have it in appendices and xref to it, possibly earlier]

## 6.2. Higher Order Functions

### 6.2.1. Passing Functions as Parameters

One very nice thing about functions, and one of the coolest and most powerful things in all of Haskell, is that functions can take functions as parameters. The simplest example (we've intentionally given the following a name that's not revealing) is this:

```
f2 :: (a -> a) -> a -> a
f2 f x = f (f x)
```

What's with the parentheses in the type declaration? They indicate that the whole `(a -> a)` thing is a single parameter: a function that takes something of a type and returns something of the same type. We need them because the `->` is right-associative — otherwise it would treat the first `a` and the second `a` as separate, single parameters[4].

On to the body of the function: `f2` takes a function, `f`, and a value, `x`. What it does is apply `f` to `x` (the `f x` part), then apply `f` again to the result. Essentially, `f2` applies a function twice. In Mathematics class we would have written something like $f^2(x) = f(f(x))$. Notice the similarity.

```
ghci> succ 3 -- successor
4
ghci> succ 4
5
ghci> succ (succ 3)
5
ghci> f2 succ 3
5
ghci> f2 pred 3 -- predecessor
1
ghci> f2 sqrt 16
2.0
ghci> f2 tail "abcd"
"cd"
ghci> f2 head "abcd" -- whoops, we need the function to return the same
   type

<interactive>:22:4:
    Couldn't match type ‘Char’ with ‘[Char]’
    Expected type: [Char] -> [Char]
      Actual type: [Char] -> Char
```

[4]We know that functions really only take a single parameter at a time. But it would save us some time and effort to think of them as taking several parameters.

```
    In the first argument of ‘f2’, namely ‘head’
    In the expression: f2 head "abcd"
```

We now understand better how `f2` works and we know why the function we pass has to have the type `(a -> a)`. If our function takes an `Int` and returns a `Bool`, there's no way we can call it again on the resulting `Bool` — it's the wrong type.

While we can call `head` twice on something like `[[2,3],[4,5]]` (it returns `2`), using `f2` will give an error. Moreover, there's no easy way to modify it so it can work.[5] We'll discuss this in [XREF], as well as provide an adequate solution.

Very few functions take a single parameter and return something of the same type. We can, however, partially apply functions to the point of accepting only one parameter, and then pass them to `f2`. It's obvious how useful partial application becomes in this case.

```
ghci> f2 (+ 2) 9
13
ghci> f2 (* 5) 4
100
ghci> f2 (^2) 3
81
ghci> f2 ("a" ++) "b"
"aab"
ghci> f2 (++ "a") "b"
"baa"
ghci> f2 ('a':) "b"
"aab"
```

So let's recap what's going on here, because it's important. `f2` looks like this:

```
f2 :: (a -> a) -> a -> a
f2 f x = f (f x)
```

Basically it applies the function `f` (of type `a -> a`) to `x` (a value of type `a`) twice. We can create a function to apply it three times, or even four:

```
f3 :: (a -> a) -> a -> a
f3 f x = f (f (f x))

f4 :: (a -> a) -> a -> a
f4 f x = f (f (f (f x)))
```

The type remains the same because we still have only two parameters: the function and the value to apply it to.

### 6.2.2. Flipping the Parameters

Sometimes we want to call a function with the parameters in another order. For instance, maybe we want to call our drink calculator (4.2.2, reproduced here for our convenience) in the order `n sex`.

```
drink :: (Fractional a, Ord a) => String -> a -> String
drink sex n
    | bac < 0.03 = "You're␣as␣sober␣as␣can␣be␣expected."
    | bac < 0.08 = "You␣can␣drive,␣but␣it's␣a␣bad␣idea."
```

[5]The problem is that `[[2,3]],[4,5]]` is a list of lists, but calling `head` on it returns a list (namely, `[2,3]`), which has a different type.

```
    | bac < 0.10 = "Your␣reasoning␣is␣out␣the␣window."
    | otherwise  = "Stop␣drinking."
    where bac = n * if sex == "male" then 0.025 else 0.035
```

We can define an additional function like below, but since we're talking about higher-order functions, there is another way.

```
flipDrink :: (Fractional a, Ord a) => a -> String -> String
flipDrink n sex = drink sex n
```

In this case, we shall use `flip`. `flip` is a nice built-in function that reverses the parameters of a two-parameter function. We can define our own version of it[6]:

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

The reasoning is pretty intuitive but can still be confusing: we want to feed the parameters in reverse order, but the function will only accept them in the right one. So we give the parameters in the wrong order (what we want) and `flip'` will call them in the right order (what the compiler wants), just like `flipDrink` above. Some examples:

```
ghci> drink "female" 2
"You␣can␣drive,␣but␣it's␣a␣bad␣idea."
ghci> flip' drink 2 "female"
"You␣can␣drive,␣but␣it's␣a␣bad␣idea."
ghci> (-) 3 2
1
ghci> flip' (-) 2 3
1
ghci> (++) "hello" "world"
"helloworld"
ghci> flip' (++) "hello" "world"
"worldhello"
ghci> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
ghci> flip' zip [1,2,3] [4,5,6]
[(4,1),(5,2),(6,3)]
ghci> flip' (zip [1,2,3] [4,5,6]) -- nope, error
```

What happens if we partially apply `flip'`? If we only give it the function, we get that function with its parameters reversed.

```
ghci> :t zip
zip :: [a] -> [b] -> [(a, b)]
ghci> :t flip' zip
flip' zip :: [b] -> [a] -> [(a, b)]
ghci> let oddDivision = flip (/)
ghci> 2 `oddDivision` 3
1.5
```

If we give it *a function* and a parameter, we've essentially partially applied *that function* on its second parameter:

[6]Quick reminder: despite what the syntax highlighter may imply [FIXME-I'm working on it, but it seems to be a particularly thorny problem], the quote doesn't do anything. It's just another character in the function name so that `flip'` won't overlap with the predefined `flip`.

```
ghci> let compare2With = compare 2
ghci> let compareWith2 = flip compare 2
ghci> compareWith2 3
GT
ghci> compare2With 3
LT
```

## 6.3. More Useful Functions

### 6.3.1. `map` and `zipWith`

Another cool (and useful) thing we can do is apply a function to every element in a list using `map`. Like before, we can have our own `map`, which we'll call `map'`.

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

This is the first time we use higher order functions and recursion simultaneously. First, as always, the type declaration: `map'` takes a function (that takes something of type `a` and returns something of type `b`) and a list of somethings of type `a` and returns a list of somethings of type `b`.[7]

Recall how we learned them during the recursion chapter. The second line is the base case: mapping a function (any function, thus the `_`) over the empty list is the empty list.

The third line: mapping a function `f` over a list with the first element `x` and the rest of the elements `xs` is a list with the first element `f x` and the rest of the elements obtained by mapping `f` over `xs`. In other words, we apply the function element by element, starting with the first one. Example:

```
ghci> map' succ [6,9,3]
[7,10,4]
```

1. `map' succ [6,9,3]` is `succ 6 : map' succ [9,3]`, which is `7 : map' succ [9,3]`
2. `map' succ [9,3]` is `succ 9 : map' succ [3]`, which is `10 : map' succ [3]`
3. `map' succ [3]` is `succ 3 : map' succ []`, which is `4 : map' succ []`
4. `map' succ []` is `[]`, so `map' succ [6,9,3]` is `7 : 10 : 4 : []`, which is `[7,10,4]`

We're gonna assume that we've gained a sufficient understanding of recursion such that elaborations like the one above aren't necessary from now on.

[FIXME] NOTE: if I haven't explained things well enough and by this point you do not *fully* understand recursion, especially with higher-order functions, shoot me an e-mail at questions@sthaskell.com telling me where you got lost so I know where to improve. I'd really appreciate it. Thanks!

Some more examples with `map'`, also highlighting some more partial application uses.

```
ghci> map' pred [6,9,3]
[5,8,2]
ghci> map' sqrt [4,9,16]
[2.0,3.0,4.0]
ghci> map' (+2) [10,20,30,40]
[12,22,32,42]
```

[7]Shorter explanation: `map'` takes a function (that takes an `a` and returns a `b`) and a list of `a`s and returns a list of `b`s.

```
ghci> map' (==5) [2,5,3,5]
[False,True,False,True]
ghci> map' (4/) [4,2,1,0.5]
[1.0,2.0,4.0,8.0]
ghci> map' (++"aa") ["bb", "cc"]
["bbaa","ccaa"]
ghci> map' ('x':) ["b", "a", "r"]
["xb","xa","xr"]
```

Another function, `zipWith`, is just like `map`, but it operates on *two* lists and takes a two-parameter function. Our own version might look something like this:

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

Again, notice how extremely similar to `map` it is. So, `zipWith` applies a two-parameter function to the elements of two lists, returning a third list with the results. It finishes when one of the lists is empty. Examples:

```
ghci> zipWith' (+) [2,3,4] [5,6,7]
[7,9,11]
ghci> zipWith' (++) ["hello␣","bye␣"] ["world","everyone"]
["hello␣world","bye␣everyone"]
ghci> zipWith' (*) [1..6] [2,2..]
[2,4,6,8,10,12]
ghci> zipWith' compare [5,6,7] [3,10,7]
[GT,LT,EQ]
ghci> zipWith' (&&) [True,True] [True,False]
[True,False]
ghci> zipWith' (++) ["aa", "bb"] ["xx", "yy"]
["aaxx","bbyy"]
```

Now we see another useful application of `flip`[8]. Not necessarily the following example, but the fact that we can pass a function with its parameters in another order.

```
ghci> zipWith' (flip (++)) ["aa", "bb"] ["xx", "yy"]
["xxaa","yybb"]
ghci> flip (zipWith' (++)) ["aa", "bb"] ["xx", "yy"]
["xxaa","yybb"]
```

It's interesting how both methods work. The first one passes a function with its parameters reversed. The second flips the lists around. The end result is the same, but we usually use the first one as it's more readable.

Remember the `zip` function back in 5.1.3? It turns out it's a specific case of `zipWith`, namely `zipWith (,)`[9].

```
ghci> zip [1,2,3] "abc"
[(1,'a'),(2,'b'),(3,'c')]
ghci> zipWith (,) [1,2,3] "abc"
[(1,'a'),(2,'b'),(3,'c')]
```

Additionally, we can continue with the `map` and `zipWith` idea and provide something that works on three lists. There actually is such a function, `zipWith3`. It looks like this:

[8] No pun intended.

[9] We've met `(,)` in 3.3.3, when discussing tuples.

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWith3 _ [] _ _ = []
zipWith3 _ _ [] _ = []
zipWith3 _ _ _ [] = []
zipWith3 f (x:xs) (y:ys) (z:zs) = f x y z : zipWith3 f xs ys zs
```

It's fairly easy to create such functions for 4, 5 or even more lists, but extremely difficult to make one to work for an arbitrary number of them. We'll look into this much later on, in [XREF].

### 6.3.2. Working with Predicates

A predicate is a function that takes a single parameter and returns a boolean (it essentially tells us if something is true). For instance, `null`, `(>3)`, `even`, `(==2)`, `or`, `elem 'a'`, and `isInfinite` are all predicates (notice how some of them are partially applied functions). They can be used as such, like below, or can be passed to a higher-order function.

```
ghci> null [2,3]
False
ghci> (>3) 6
True
ghci> even 5
False
ghci> (==2) 2
True
ghci> or [True, True, False]
True
ghci> (elem 'a') "hello␣world"
False
ghci> isInfinite (1/0)
True
```

Using them as parameters for other functions can be extremely useful, but first we need to know a couple of functions that accept predicates. `filter` is one of them — it takes a predicate and a list and returns a list containing only the elements that satisfy the predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs
                         else     filter p xs
```

We immediately notice the predicate: it's the first parameter, of type `a -> Bool`. The function traverses the list, element by element, keeping those that satisfy the predicate `p`[10] and excluding those that don't (`if p x` then include else exclude).

```
ghci> filter even [5,4,2,1,3,6,8,2]
[4,2,6,8,2]
ghci> filter (>3) [4,3,2,1,5,0]
[4,5]
ghci> filter (/= 5) [4,5,6,7]
[4,6,7]
ghci> filter (elem 'a') ["hello", "abstract", "gemini"]
["abstract"]
```

[10]While we call our functions `f`, `g` and so on, we usually name predicates `p` and `q`.

```
ghci> filter null [[5,6],[7],[],[8,9],[]]
[[],[]]
```

Even better, we can incorporate `filter` into bigger functions that do useful things — like `quicksort`.

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = lesserSorted ++ [x] ++ greaterSorted
    where lesserSorted  = quicksort (filter (<= x) xs)
          greaterSorted = quicksort (filter (>  x) xs)
```

We've recycled the example from 5.3.2, but instead of using list comprehensions, we used filters. In fact, more of the stuff we've discussed so far (like `map`) have a list comprehension equivalent. We'll talk more about this in 6.3.3.

Before we discuss applications, let's look at two functions which are very similar to `filter`: `takeWhile` and `dropWhile`.

- `takeWhile` takes a predicate and a list. Like `filter`, it takes elements which satisfy the predicate. Unlike `filter`, it stops entirely when it encounters an element that doesn't satisfy.
- `dropWhile` is similar to `takeWhile` — but it returns the rest of the list, starting with the first element that doesn't satisfy.

```
ghci> filter (>3) [4,6,2,1,8,7]
[4,6,8,7]
ghci> takeWhile (>3) [4,6,2,1,8,7]
[4,6]
ghci> dropWhile (>3) [4,6,2,1,8,7]
[2,1,8,7]
ghci> filter (/= '␣') "hello␣dear␣world"
"hellodearworld"
ghci> takeWhile (/= '␣') "hello␣dear␣world"
"hello"
ghci> dropWhile (/= '␣') "hello␣dear␣world"
"␣dear␣world"
```

We'll let the source code speak for itself (this time we're using guards instead of explicit if..else, and we're showing another indentation style[11]):

```
takeWhile                  :: (a -> Bool) -> [a] -> [a]
takeWhile _ []             =  []
takeWhile p (x:xs)
             | p x         =  x : takeWhile p xs
             | otherwise   =  []

dropWhile                  :: (a -> Bool) -> [a] -> [a]
dropWhile _ []             =  []
dropWhile p xs@(x:xs')
             | p x         =  dropWhile p xs'
             | otherwise   =  xs
```

Recall the "as patterns" (4.1.4): `name@pattern` allows us to reference `pattern` by using `name`; in our case, `name` is `xs` and `pattern` is `(x:xs')`.

[11]These examples are identical to those in the official source code. It's not a coincidence; that's where I took them from.

### 6.3.3. Comparison with List Comprehensions

Some higher-order functions that operate on lists, namely `map` and `filter`, are equivalent to using list comprehensions. We can even define them this way:

```
map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
```

Should we use list comprehensions or higher-order functions? Usually we use the former when we have multiple operations to perform and the latter otherwise. For instance, `[ 2*x | x <- xs, even x, x >= 2 ]` can be expressed by nesting maps and filters, like `map (2*) (filter even (filter (>=2) xs))`, but is extremely unreadable. Conversely, `map (+2) xs` is much more concise than `[ x + 2 | x <- xs ]`.

One extremely cool thing that can be done with `map` is creating a list of functions — by passing a two- (or more-) parameter function, such as `*`.

This means that the resulting list will contain partially applied functions: `(5*)`, `(4*)` etc. We can extract elements from it and fully apply them: [FIXME-elaborate on this]

```
ghci> let functions = map (*) [5,4,3,2,6]
ghci> :t functions
functions :: [Integer -> Integer]
ghci> (head functions) 8
40
```

We can totally do it with list comprehensions, as well:

```
ghci> let functions = [ (x*) | x <- [5,4,3,2,6] ]
ghci> (functions !! 4) 2
12
```

Psst! The `!!` function begins numbering at `0`. So while `6` is the *fifth* element, we need to use `!! 4`. Performing `!! 5` will result in an `index too large` error.

`takeWhile` and `dropWhile` don't have an easy list comprehension equivalent, so we won't talk about them here. Instead, we'll discuss the difference between the following:

- `zipWith (+) [1,2,3] [10,20,30]`
- `[ x + y | x <- [1,2,3], y <- [10,20,30] ]`

While `zipWith` combines corresponding elements of the list (`1` with `10`, `2` with `20` and `3` with `30`), the list comprehension matches all possible combinations (`1` with `10`, `1` with `20`, `1` with `30`, `2` with `10` and so on).

```
ghci> zipWith (+) [1,2,3] [10,20,30]
[11,22,33]
ghci> [ x + y | x <- [1,2,3], y <- [10,20,30] ]
[11,21,31,12,22,32,13,23,33]
```

It's a fundamental difference, but also one easily overlooked.

***Warning!*** Do not confuse `zipWith` with similar list comprehensions.

### 6.3.4. Anonymous Functions (Lambdas)

We've already encountered some functions which needed to be used only once. Initially we separately defined them. Afterwards, we defined them inside a `let` or a `where`. But what if our function is so trivial, that we'd rather not name it at all? Introducing anonymous functions, or lambdas for short. What better way to show them than to give a few examples?

```
-- Syntax: lambdas

\x -> x + 2

\xs -> length xs > 100

\x y z -> x + y + z
```

Defining anonymous functions is similar to defining regular functions, but instead of the function's name we use \[12], and instead of = we write ->.

Additionally, by using lambdas, we not only specify the function, but we also "call" it. This is a really nice timesaver, because we usually create anonymous functions to pass them to higher-order functions, where they will be "called" anyway[13]. Compare the two:

```
ghci> let f x = 2*x + 3
ghci> f 5
13
ghci> (\x -> 2*x + 3) 5
13
```

Notice how we put the lambda in parentheses. Without parentheses, lambdas extend all the way to the right.

Let's see some lambdas in use. They are, technically speaking, expressions, so we can fit them anywhere (where a function is needed):

```
ghci> map (\x -> 2*x + 3) [1..5]
[5,7,9,11,13]
ghci> filter (\x -> x^2 > 16) [10,20,5,4,1,6]
[10,20,5,6]
ghci> zipWith (\x y -> x + 2*y) [1,2,3] [4,5,6]
[9,12,15]
```

Don't become overzealous with lambdas, though. We might be tempted to use them when it's not necessary:

```
ghci> map (\x -> x + 2) [1,2,3]
[3,4,5]
ghci> map (\x -> sqrt x) [4,9,25]
[2.0,3.0,5.0]
```

Here, we're better off using the functions directly:

```
ghci> map (+2) [1,2,3]
[3,4,5]
ghci> map sqrt [4,9,25]
[2.0,3.0,5.0]
```

One great thing about anonymous functions is that, like regular (named) functions, we can use pattern matching in them. Unlike regular functions, though, we have only one "body" so we can use only one pattern. If that fails, crash!

```
ghci> map (\(x,y) -> compare x y) [(3,4), (5,6), (7,7), (9,8)]
[LT,LT,EQ,GT]
ghci> map (\(x:xs) -> (x,xs)) [[2,3,4], [8,10,20]]
```

[12]Because not everyone has $\lambda$ on their keyboards (I do!).
[13]It's a cheesy explanation; we should look at the examples instead.

```
[(2,[3,4]),(8,[10,20])]
ghci> map (\('a':xs) -> xs) ["animal", "anonymous"]
["nimal","nonymous"]
ghci> map (\(3:xs) -> xs) [[4,5]]
[*** Exception: <interactive>:74:6-18: Non-exhaustive patterns in lambda
```

One final cool thing before we finish with lambdas: because of currying (and the fact that lambdas extend all the way to the right if we don't put them in parentheses), the following two are equivalent:

- `\x y -> x + y`
- `\x -> \y -> x + y`

One additional consequence of currying is that we can also define functions using lambdas, but it's usually not as readable. Notice how the parameters can be moved to the right, after the `=`:

```
f2 :: (a -> a) -> a -> a
f2 f x = f (f x)

g2 :: (a -> a) -> a -> a
g2 f = \x -> f (f x)

h2 :: (a -> a) -> a -> a
h2 = \f x -> f (f x)
```

We won't focus as much on anonymous functions here because we'll use them extensively in the chapters that follow.

# 7. Folds and Scans

I see a lot of stuff with very clever maps and folds... It's like functional spaghetti code.

*(sproingie)*

## 7.1. An Introduction to Folds

### 7.1.1. Eating a List

Remember the discussion in 5.3.3 about common patterns in recursion? Here are the examples again:

```
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

and []     = True
and (x:xs) = x && and xs

or []     = False
or (x:xs) = x || or xs
```

The common pattern is:

```
listFunction []     = startingValue
listFunction (x:xs) = x `baseFunction` listFunction xs
```

Or, if we don't call it infix[1]:

```
listFunction []     = startingValue
listFunction (x:xs) = baseFunction x (listFunction xs)
```

Seeing how often we use something like this, it's natural to make a function that covers all possible use cases. As we've discussed earlier, creating a more abstract, general function that can be reused in many different ways is at the heart of Haskell. Such a thing, however, would be impossible without knowing about higher-order functions (which we now do).

Let's think about what we'll need. We obviously would want to have our base function, but also provide the starting value and a list on which to perform the operations. Here they are:

1. `baseFunction`, which takes two parameters and returns a third. This could be any of the following: `+`, `*`, `&&` etc.
2. `startingValue`, which can be `0`, `1`, `True` or any other value
3. `xs`, the list on which to perform the operations.

[1]Refresher: A prefix function comes before its parameters: `f x y`. An infix function is between them: ``x `f` y``. The notations are equivalent. Notice the backquotes.

Let's call our function `eat`[2], because it kinda looks like we're eating the list element by element.

First of all let's think about the type definition. It takes `baseFunction` (which can be something like `a -> b -> c`), a starting value (let's say `d` to keep it general), a list of some type `[e]`, and finally, it returns something, let's say of type `f`. It would look something like `eat :: (a -> b -> c) -> d -> [e] -> f`.

But there's something wrong with our type definition. If our list is of type `[e]`, then we need the base function to call elements of that type as well, or it won't work on our list. It should look more like `eat :: (e -> e -> e) -> e -> [e] -> e`. This one, though, looks a bit too specific. We want our functions to be as general as possible and certainly a function that only takes values of the same type can be improved[3].

Let's skip this one for the moment and move on to actually defining the function. Let's call `someFunction` `f` and `startingValue` `z`[4] because it's shorter and easier to follow. The edge condition should be pretty easy — eating an empty list should give us the starting value[5]. Let's write this down.

```haskell
eat _ z [] = z
```

Notice how there's an underscore in there. That actually represents the base function `f`, but since we don't need it here, we write `_`.

Next up, doing the actual function. First of all, we'll separate the list into a head and a tail, because that's what all the functions above have done. Our function is an *abstraction* of those, so it should behave the same.

```haskell
eat _ z [] = z
eat f z (x:xs) =
```

Similarly, `f` should be called with the head of the list, `x`, and something else.

```haskell
eat _ z [] = z
eat f z (x:xs) = f x
```

Now, what is `f`'s second parameter? The whole thing should be recursive, so the logical choice would be to call `eat` with `xs`. This way we'll be sure to follow the example of `sum`, `product` etc. above. We shouldn't forget the parentheses to group `eat xs`.

```haskell
eat _ z [] = z
eat f z (x:xs) = f x (eat xs)
```

Wait! We forgot to carry the other parameters that `eat` needs: the base function `f`, the starting value `z` (which will be used when `eat` finally reaches the empty list), and obviously the tail of the list, `xs`. Those weren't needed for `sum` or `product` but we need them now. There:

```haskell
eat _ z [] = z
eat f z (x:xs) = f x (eat f z xs)
```

Let's put it in a file (say, `eat.hs`) and load it.

```
ghci> :l eat.hs
[1 of 1] Compiling Main             ( eat.hs, interpreted )
Ok, modules loaded: Main.
```

[2]A better idea might have been `traverse`, but there's already a function with that name, and it does something entirely different (see [XREF]).

[3]In most cases, anyway.

[4]We could be tempted to call it `x0`, but someone skimming over the function might get confused and think that `x` (which we'll use for the elements of the list) and `x0` have the same type because they're named similarly. We don't know if that's the case!

[5]If we call `product`with `[]` it returns `1`, `sum [] = 0`, `and [] = True` etc.

It compiled! Now for the thing that we skipped earlier: the type.

```
ghci> :t eat
eat :: (t -> t1 -> t1) -> t1 -> [t] -> t1
```

It seems that our `eat :: (a -> b -> c) -> d -> [e] -> f` guess was indeed too broad, and our next best guess, `(e -> e -> e) -> e -> [e] -> e` was too specific. But we were close! Let's write the type declaration (using `a` and `b` instead of the ugly `t` and `t1`), align things a little and marvel at our handiwork:

```
-- File: eat.hs
eat :: (a -> b -> b) -> b -> [a] -> b
eat _ z []     = z
eat f z (x:xs) = f x (eat f z xs)
```

Now let's go ahead and try to define `sum` in terms of `eat`. Our function is addition, `+`. The starting value is `0` (because 0 doesn't influence addition, i.e. `0 + anything` gives that thing back). Let's go!

```
sum' xs = eat (+) 0 xs
```

The `xs` is redundant[6], so we can remove it.

```
sum' = eat (+) 0
```

We can try it out (`sum'` uses `eat` so we should be sure that `eat` is defined in the same file[7]!) and see if it works properly.

```
ghci> sum' [1,2,3,4,5]
15
ghci> sum' []
0
```

Excellent! We can go ahead and define the other functions in here just as easily. Notice how, every time we choose a starting value, we try to start with something that doesn't influence the result: `1 * anything`, `True && anything` and so on. They all give that thing back[8].

```
product' = eat (*) 1
and' = eat (&&) True
or' = eat (||) False
```

Let's test them as well.

```
ghci> product' [4,5,6]
120
ghci> product' [0]
0
ghci> and' [True, True, False]
False
ghci> or' [False, True, False]
True
```

This `eat` function is really useful. How come it's not predefined? Let's do a Hoogle search for its type and see if there's a similar function included with Haskell.

Whoops! It found something: `foldr`. It looks like we've just reinvented the wheel. It has the same type, it appears to do the same thing, and it might just be the same function! Let's check it out.

[6]It's because of the fundamentals of currying and partial application that we've discussed earlier in 6.1.1)
[7]Quick note: Haskell can include a file in another file so we can have two interdependent functions in different files. We'll learn how when we do modules, in [XREF].
[8]Mathematically, they're called *identity elements*. Wikipedia has a neat list of examples for lots of functions.

```
ghci> let sum' = foldr (+) 0
ghci> sum' [1,2,3,4,5]
15
```

So this function has already been implemented. This is both a blessing and a curse with Haskell — on one hand, a lot of the things that we might need in a program are already there, ready to be used. But on the other hand, when writing a larger program, we're bound to code a function that has already been implemented more efficiently.

Actually, `foldr` is a great example of this. The way we've written `eat` is the "academic" way — the function is easy to understand, concise and it's the proper mathematical definition. The official definition of `foldr` is longer and harder to understand — but it's more efficient.

As an off-topic note, we're gonna have to be careful when we write our code in Haskell: we want something short and easy to understand, but we also want something that doesn't take 100 years to run. It's like this[9]:

```
--- BEAUTIFUL <------------------->  <-------------------> EFFICIENT ---
* easy to understand             WHAT                  runs faster *
* concise, elegant, fun           WE              uses less memory *
* harder to make mistakes        WANT       good for large programs *
```

## 7.1.2. Introducing Folds Proper

Now that we've played around with `eat`, which turned out to be our own implementation of `foldr`, it's time to learn properly about folds and how they are truly useful. We should note that `foldr` (right fold) is one of the two big types of folds out there, the other one being `foldl` (left fold). We'll stick with `foldr` for the time being.

Let's pull up our definition of `foldr` again and mention some of the terms people use when referring to folds.

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

- `foldr` is called a "right fold". It's counterintuitive, but `foldr` actually eats the list from the right. We'll get back to this really soon, in [XREF].
- `f` is called the "accumulating function", the "combining function" or just "the function".
- `z` is called the "accumulator". It's the value that gets built up ("accumulated") and eventually returned by the fold.
- `(x:xs)` is the "list". It's the thing that gets "folded" by `foldr`.

With terminology out of the way, now it's time to discuss one of the most important things we need to understand in order to best use `foldr`: how the accumulator gets built up and retuned.

A quick way to try and understand folds is by looking at what happens with a simple example:

```
ghci> foldr (+) 0 [1,2]
3
```

1. `foldr (+) 0 [1,2]` is `1 + (foldr (+) 0 [2])`
2. `foldr (+) 0 [2]` is `2 + (foldr (+) 0 [])`, so `foldr (+) 0 [1,2]` is `1 + (2 + (foldr (+) 0 []))`

[9] I strongly believe that a Haskell program should in some manner be beautiful and concise. That's why we use it. If we want to write the most efficient program out there, we're probably gonna use a different language.

3. `foldr (+) 0 []` is `0`, so `foldr (+) 0 [1,2]` is `1 + (2 + 0)`, which is `1 + 2`, which is `3`

We can see that `foldr` goes through the list beginning with the first element, but it puts all the operations on hold[10] until it processes the last element, getting to the empty list. When it gets to the empty list, it returns the starting value of the accumulator (`0`). That value gets passed to the combining function (`+`) along with the last element (`2`) which yields `2 + 0`, which returns a new value for the accumulator, which gets passed to the combining function along with the previous element, and so on.

More concisely, `foldr` calls the function `(+)` with the last element (`2`) and the starting value of the accumulator (`0`), obtaining a new accumulator value. The function is now called with the second-to-last element and the *new* accumulator value, yielding an even newer accumulator. The process gets repeated until `foldr` has gone through the entire list. The final accumulator value is what gets returned by `foldr`.

Let's go through a more complicated example using our more concise method[11]:

```
ghci> foldr (/) 2 [5,6,3,4]
1.25
```

1. The initial accumulator value is 2. The function `/` gets applied to the last element, `4`, and the accumulator. `4/2` is `2`, which is the new accumulator value[12].
2. Now `/` gets applied to `3` (the second-to-last element) and `2`, which is the current value of the accumulator. `3/2` is `1.5`, which is the new accumulator value.
3. `6/1.5` is `4`, which is the new accumulator value.
4. `5/4` is `1.25`. Because `foldr` has now gone through the entire list, `1.25` is what gets returned in the end.

It's important to note that even though `f` is a two-parameter function, it does *not* operate on two elements of the list. The combining function operates on a *single* element of the list at a time — the accumulator is its other parameter. Let's look at a slightly different example:

```
ghci> foldr (\x acc -> even x && acc) True [4,6,7,8]
False
```

Wow, that's a mouthful! Let's understand this piece of code by breaking it into chunks:

- `foldr` is there, so we should expect three parameters: a function, a value, and a list.
- `\x acc -> even x && acc` is the accumulating function. We'll mainly focus on this one, as it's the function that does the magic.
- `True` is the starting value of the accumulator. It might get changed when it's passed around.
- `[4,6,8,11]` is the list that will get folded.

So what's up with `\x acc -> even x && acc`? This is an anonymous function, as we've discussed in 6.3.4. It takes two parameters, `x` and `acc`[13]. This is a tricky one because `x` is an integer, but the accumulator is a boolean.[14] It's okay for the accumulator to have a different type, as long as the combining function returns something that has the same type as the accumulator.

Let's try the combining function with some arbitrary values, and then attempt to see what happens during the fold.

[10]In `1 + (2 + ...))`, `2` is inside a pair of parentheses, so Haskell can't do `1 + 2` yet.

[11]We don't usually use functions like division with folds, because there are different types of folds, and division gives different results if we start from the left or from the right.

[12]If you come from a more traditional language like C or Python, remember that the "accumulator" is not a variable that changes value. At every step, the function takes an accumulator and returns a new one.

[13]We usually call the accumulator `acc` for brevity and ease of understanding.

[14]We can deduce this in many different ways: we can look at `even x && acc` to guess the types, plug the anonymous function into GHCi with `:t`, or glance at `True` and `[4,6,7,8]`, our intitial values which get passed into `foldr`.

```
ghci> let f = (\x acc -> even x && acc)
ghci> f 2 True
True
ghci> f 2 False
False
ghci> f 3 True
False
ghci> f 3 False
False
```

1. `foldr (\x acc -> even x && acc) True [4,6,7,8]` has a starting accumulator of `True`
2. `even 8 && True` is `True && True`, which is `True`, our new accumulator.
3. `even 7 && True` is `False && True`, which is `False`.
4. `even 6 && False` is `True && False`, which is `False`.
5. `even 4 && False` is `False`, which is our final accumulator value — this is what `foldr` returns.

We can approach this in a more intuitive fashion as well — the `&& acc` makes it such that once the accumulator value becomes `False`, it will remain `False`. The accumulator value becomes `False` when it reaches an odd element (`even x` would be false, and `False && anything` is `False`). The accumulator starts out `True`, therefore we can intuitively guess that `foldr` will return `False` if there's at least one odd element in the list, and `True` otherwise. This is much less effort but it's important to check our intuition!

```
ghci> foldr (\x acc -> even x && acc) True [2,4,6,8]
True
ghci> foldr (\x acc -> even x && acc) True [2,4,6,9]
False
```

Our intuition is, in fact, correct.

As long as we think logically about what happens during a folding operation, whether by expanding the recursion and following the calculations (our first approach), by using the more concise method of following the accumulator through the list from the last element to the first (our second approach) or by simply thinking about it intuitively (our third approach), we won't have any surprises when we do more complicated things with folds.

The more we go through folds and use them, the less we'll need to use the step-by-step approach and the more natural folding will seem to us.

### 7.1.3. When You Should Fold

Now that we know what folds do and how they work, our next logical question is when we should use them. The answer is natural: we should use them when they fit, namely when we need something to go through a list, element by element, and return a result. We've already seen addition, multiplication, and so on. Let's think of other use cases.

Suppose we have a list of positive numbers, and we're asked to find out the maximum of these values. To do that, we go through the list and compare each element with the current accumulator, keeping the larger one. After we go through the entire list, we should get the maximum value. Let's get going!

First of all, we need a function that takes the maximum of two things: our element, and the accumulator.

```
ghci> max 5 7
7
ghci> max 13.2 11
13.2
```

Next, we need to pass it into `foldr`, along with our list and a starting value.

```
ghci> let myList = [5,7,10,2,3]
ghci> foldr max startingValue myList -- What should we put here?
```

We know that the list contains only positive numbers, so if we put 0 as the starting value, it shouldn't influence the eventual result: all the numbers in the list are greater than 0.

```
ghci> foldr max 0 myList
10
```

As we did before, we can save this in a separate file, and also write a type definition for it (`Num` is required by `0` and `Ord` is required by the use of `max`):

```
-- File: maximum-positive.hs
maximumPositive :: (Num a, Ord a) => [a] -> a
maximumPositive = foldr max 0
```

We can also load this and test it out on more lists:

```
ghci> :l maximum-positive.hs
[1 of 1] Compiling Main             ( maximum-positive.hs, interpreted )
Ok, modules loaded: Main.
ghci> maximumPositive [3, 6, 2]
6
ghci> maximumPositive [7.3, 6.5]
7.3
```

It works! We've deliberately avoided using negative numbers because our initial problems specifies that all numbers are positive. We'll return to this example shortly, trying out negative numbers as well, in [XREF].

Here's a more practical example. Imagine we have a list of bank accounts that contain the account holder's name and their balance, something like `("Steve", 150.32)`. We need to see how many accounts in the list have a negative balance. To do this, we need a function that takes an element and checks to see if the balance is less than zero. If it is, we should return the accumulator plus one. This way, the accumulator will end up being the number of accounts that are in the red.

Let's begin by working out the function that does the comparison. We're going to use an anonymous function. It should look something like `\x acc -> if <account is in red> then acc + 1 else acc`. In this case, the balance is stored in the second value of the tuple, so we can use `snd x < 0` to check whether the balance is negative. Our function is: `\x acc -> if snd x < 0 then acc + 1 else acc`. Let's try it out:

```
ghci> (\x acc -> if snd x < 0 then acc + 1 else acc) ("Mary", 140.3) 5
5
ghci> (\x acc -> if snd x < 0 then acc + 1 else acc) ("John", -120.5) 5
6
```

We see that it incremented the accumulator for John's balance, but not Mary's, which means we're on the right track. Now all we need to do is actually pass this into `foldr`. We're starting the counting at 0.

```
ghci> let negAccts = foldr (\x acc -> if snd x < 0 then acc + 1 else acc) 0
ghci> negAccts [("Steve", 142.5), ("Mary", -230.2), ("Sarah", 1500.0)]

<interactive>:31:21:
    No instance for (Fractional Integer)
      arising from the literal ‘142.5’
    Possible fix: add an instance declaration for (Fractional Integer)
    In the expression: 142.5
    In the expression: ("Steve", 142.5)
    In the first argument of ‘negAccts’, namely
      ‘[("Steve", 142.5), ("Mary", - 230.2), ("Sarah", 1500.0)]’
```

Whoa, what's going on? Let's read this error[15]: it seems that it's trying to turn an `Integer` into a `Fractional` value because it encountered `142.5` when we called `negAccts`. In this case, it seems that our use of `0` is problematic. When we created the function `negAccts`, Haskell automatically tried to infer its type (we didn't provide an explicit type declaration). Because we used `0` instead of `0.0`, it assumed we are talking about `Integers` here, when we actually want to use any numbers that work, including `Fractionals`. This is only one of the possible ways that Haskell can infer types incorrectly.

Fortunately, we can fix this in several different ways. While we can replace `0` with `0.0` in `snd x < 0` and make the function work just fine, or alternatively use integer values for the bank account balances, we should do the "right" thing and place the function inside a file, along with a type declaration:

```haskell
-- File: neg-accts.hs
negAccts :: (Num a, Ord a) => [(String, a)] -> Int
negAccts = foldr (\x acc -> if snd x < 0 then acc + 1 else acc) 0
```

Now it works:

```
ghci> :l neg-accts.hs
[1 of 1] Compiling Main             ( neg-accts.hs, interpreted )
Ok, modules loaded: Main.
ghci> negAccts [("Steve", 142.5), ("Mary", -230.2), ("Sarah", 1500.0)]
1
```

Let's move on to another example. This is a bit different from the other ones. In this scenario, we're trying to determine if all the elements in a list are in ascending order. If our function is called `isAscending`, we'd expect something like this:

```
ghci> isAscending ['a', 'b', 'd', 'f', 'm', 'q', 'r']
True
ghci> isAscending [6, 7, 4, 8, 9]
False
```

Our function doesn't seem to work that well with folds — checking if a list is ascending requires comparing two elements with each other at every step, and folds only operate on one element at a time. It appears that there's no easy way to write this in terms of folds.

Let's think of an implementation without folds first, in order to make sure that we didn't miss anything. `isAscending` should take a list of comparable elements and return a boolean (`True` or `False`).

We'd ideally compare elements two by two. If the first one is larger than the second, then the list is not ascending and we return `False`. Otherwise, we compare the next elements to see if they are ascending, and so on until we reach the empty list or a list with only one element, which are ascending (so we'll return `True`).

First things first: the type definition, and the base cases.

```haskell
-- File: is-ascending.hs
isAscending :: (Ord a) => [a] -> Bool
isAscending []  = True
isAscending [_] = True
-- to be continued...
```

[15]A super detailed description on how to read errors is in B.2.1.

So far, so good. Now we need to separate the list into the first two elements and the rest of the list, and tackle the case when they are not in ascending order (the easier one). If the first two elements are in order, we need to check if the rest of the list is ascending as well:

```haskell
-- File: is-ascending.hs
isAscending :: (Ord a) => [a] -> Bool
isAscending []  = True
isAscending [_] = True
isAscending x:y:ys
    | x > y     = False
    | otherwise = isAscending ys
```

It looks finished! Let's try it out:

```
ghci> :l is-ascending.hs
[1 of 1] Compiling Main             ( is-ascending.hs, interpreted )
Ok, modules loaded: Main.
ghci> isAscending [1,2,3,4,5]
True
ghci> isAscending [2,1,3,4,5]
False
ghci> isAscending [1,3,2,4,5]
True
```

Wait, something's wrong. Why does it return `True` for the third one? If we look back at the code, we see that it compares the first two elements, which is good. But then it jumps straight to comparing the third and the fourth, without checking if the second and the third are in order.

In other words, our function only checks every other comparison. In the `otherwise` guard, we need to have `isAscending (y:ys)` instead of `isAscending ys`. This way it won't skip any comparisons.

```haskell
-- File: is-ascending.hs
isAscending :: (Ord a) => [a] -> Bool
isAscending []  = True
isAscending [_] = True
isAscending (x:y:ys)
    | x > y     = False
    | otherwise = isAscending (y:ys)
```

It's good we caught that mistake early! If we only stopped at the first two tests, we probably wouldn't have noticed it as quickly. Therefore, it is important to perform lots of tests on our code, especially if it's part of a large program.

```
ghci> :r -- Reload the loaded files
Ok, modules loaded: Main.
ghci> isAscending [1,3,2,4,5]
False
ghci> isAscending ['a','b','c']
True
```

Now let's go back and try to implement this as a fold. Looking at the definition of `foldr` below, it becomes increasingly clear that `isAscending` doesn't follow the same pattern.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- `isAscending` operates on two elements at once, whereas `foldr` goes element by element.
- `isAscending` has two base cases, whereas `foldr` has only one.
- `isAscending` contains a guard in one of the cases, whereas `foldr` just recurses directly, without any conditionals.

For all these reasons, there's no straightforward way to implement `isAscending` by using `foldr`.[16] It's okay, though. Sometimes even something as flexible as `foldr` isn't particularly suited to a certain problem. As we progress through the book, we'll have a wider toolset to deal with specific tasks, but we'll still return to basics such as recursion every once in a while — in Haskell, the basics are very powerful.

## 7.2. Different types of folds

### 7.2.1. `foldl` vs `foldr`

We've talked at length about `foldr` and the way it works. The other major type of fold is the left fold. In the case of left folds, the list gets eaten up from the left. Let's compare `foldl` and `foldr`'s definitions[17]:

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

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

Here, `foldr` applies the combining function to the first element and the accumulator resulted from folding the rest of the list. By contrast, `foldl` immediately applies the combining function with the first element and the initial accumulator and then it recurses into folding the rest of the list.

The difference is subtle, but very important: `foldr` goes through the list stacking up operations [FIXME-double triple check if this is correct], and when it reaches the end[18] it starts processing them beginning with the last one (essentially from right to left), whereas `foldl` goes through the list queuing up operations, and when it reaches the end it starts processing them beginning with the first one (essentially from left to right).

Let's work on a few simple examples before we discuss the details of how `foldl` and `foldr` work, and when to use each one.

Because of how `foldl` and `foldr` are defined (and by looking at the type declaration), we can see that the combining function has its arguments flipped in the case of `foldl`. This is usually not a problem, as a lot of the functions that we use are commutative (`1 + 2` is the same as `2 + 1`):

```
ghci> foldr (+) 0 [1..5]
15
ghci> foldl (+) 0 [1..5]
15
ghci> foldr (&&) True [True, True, False]
False
ghci> foldl (&&) True [True, True, False]
False
```

[16]If we employ particularly ugly hacks, such as making the accumulator store two values in a tuple, and using conditionals to replicate the behavior of the guards in `isAscending`, we might just make it work. We'll see in [XREF] how this is possible and why it's a bad idea.

[17]In GHC, `foldl` is actually implemented in a different way, for efficiency. Our definition is equivalent — we're using it to illustrate the difference between `foldl` and `foldr`.

[18]Not quite — thanks to laziness, `foldr` won't go through the entire list if it doesn't need to. We'll go back to this shortly.

Let's recycle one of our previous examples:

```
ghci> foldr (\x acc -> even x && acc) True [2,4,6,9]
False
```

We need to flip the arguments in order to make it work with `foldl`:

```
ghci> foldl (\acc x -> even x && acc) True [2,4,6,9]
False
```

[FIXME]

# Part III.

# Appendices

# A. Miscellaneous

## A.1. Functions

### A.1.1. Fixity

The following table[1] shows the precedence and fixity (left-, non-, and right- associativity) of the operators in Prelude.

| Precedence | Left-associative | Non-associative | Right-associative |
|---|---|---|---|
| 9 | `!!` | | `.` |
| 8 | | | `^`, `^^`, `**` |
| 7 | `*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` `` | | |
| 6 | `+`, `-` | | |
| 5 | | | `:`, `++` |
| 4 | | `==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` ``, `` `notElem` `` | |
| 3 | | | `&&` |
| 2 | | | `\|\|` |
| 1 | `>>`, `>>=` | | |
| 0 | | | `$`, `$!`, `` `seq` `` |

Below are some examples of precedence and fixity declarations (if an operator definition lacks a fixity declaration it is assumed to be `infixl 9`).

```haskell
-- File: fixity.hs
x ++++ y = x + y + x*y
infixl 3 ++++ -- left-associative

x -.- y = x^3 + y^3
infixr 5 -.- -- right-associative

func a b = a + b + b
infix 2 `func` -- non-associative
```

In many cases the correct fixity declaration carries a great deal of importance — let's take `-.-` (the one declared above) as an example.

```
ghci> (3 -.- 4) -.- 5
753696
ghci> 3 -.- (4 -.- 5)
6751296
```

So, even though `-.-` is right-associative in Haskell-speak, it is non-associative in the mathematical sense: `(a -.- b) -.- c` is *not* the same as `a -.- (b -.- c)`.

[1]Taken from the Haskell 98 Report

### A.1.2. Laziness Explained

Haskell has a very strange property when compared to your "usual" programming languages: it's lazy. This means that the compiler or interpreter will evaluate an expression only when it's needed. It's very tricky on many levels, mainly because laziness introduces important differences between superficially similar expressions. Let's take the simplest function imaginable and move on from there: `&&`. As a reminder, `&&` is defined like so:

```haskell
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Note how `&&` won't even evaluate its second argument if the first is `False`: `&&` is *strict* in the first argument, but *lazy* in the second. In other words, `&&` must always evaluate the first argument, but not necessarily the second one.

We have a better perspective when we look at expressions in light of *thunks*. Thunks are unevaluated values (with "instructions" on how to evaluate them)[2]. Let's take the following piece of code as an example:

```haskell
-- File: thunks.hs

a         = (length "hello", [1, 2, 3, 4])
(b, c)    = a
1:d       = c
```

Line-by-line:

1. Haskell matches `(length "hello", [1, 2, 3, 4])` to `a`. Because we do nothing to `a`, Haskell doesn't care what it is. It doesn't actually evaluate it, so `a` is just a thunk.
2. `a` will need to be matched to a pair. In order to make sure the match succeeds and to assign the necessary variables, `a` is evaluated to something like `(thunk, thunk)`. `b` and `c` become thunks.
3. `c`, previously a thunk, is now evaluated to make sure it conforms to `1:d`. `c` now "becomes" `1:thunk`.

Let's take another example: Let's try to fully evaluate `("hi", [4, 5])`[3]. The steps are as follows:

```haskell
-- Evaluation steps
thunk -- unevaluated
(thunk, thunk)
('h':thunk, thunk)
('h':'i':thunk, thunk)
('h':'i':[], thunk)
('h':'i':[], 4:thunk)
('h':'i':[], 4:5:thunk)
('h':'i':[], 4:5:[])
```

Partially evaluated values are in something called *weak head normal form*. Fully evaluated things are in *normal form*.

We don't always know which functions are strict and which are lazy, but we can check by calling them with `undefined`. If `undefined` is not evaluated (i.e. remains a thunk), nothing happens. If it is, it throws an error.

```
ghci> False && undefined
False
```

[2]In fact, if Haskell weren't lazy, there would be no such thing as a thunk: all expressions and values would always be fully evaluated.
[3]For example, by printing it — printing "forces" evaluation.

```
ghci> undefined && False
*** Exception: Prelude.undefined
```

This is our confirmation of the above: `&&` is indeed lazy in the second argument, but strict in the first.

There are some catches however. For instance, we might expect `0 * x` to not need to evaluate `x`. After all, `0` multiplied by anything is `0`[4]. So we can put `undefined`, can't we?

```
ghci> 0 * undefined
*** Exception: Prelude.undefined
```

Surprise surprise! It seems that multiplication is strict in both parameters, even when supplied `0`.

What isn't surprising is that laziness is a touchy subject — the best way to learn it is through experience.

## A.2. Constants (A.K.A. "Variables")

### A.2.1. Local "Variables"

Let's look at the following example:

```
f x y
    | g x y < 5   = "Less than 5"
    | g x y == 5  = "Equal to 5"
    | otherwise   = "Greater than 5"
    where g x y = 2*x + 3*y
```

The names ("variables") `x` and `y` appear 5 times each. Let's count them:

1. `f x y`: the parameters in `f`'s function definition
2. `g x y < 5`: the parameters used in a call to `g`.
3. `g x y == 5`: the same
4. `g x y = ...`: the parameters in `g`'s function definition (before the `=`)
5. `2*x + 3*y`: the parameters in `g`'s body (after the `=`)

While we've used the same names in all `5` instances, they are logically different. We can separate them as follows:

- Pertaining to `f`: 1, 2, and 3
- Pertaining to `g`: 4 and 5

The names pertaining to `f` are logically different than those pertaining to `g` — they seem to have the same name (to us) but internally they are different. In other languages, those pertaining to `g` would be called "local variables" because they are logically different and the difference occurs only in a limited area: `g x y = 2*x + 3*y`.

We can reflect the difference in meaning ourselves, by renaming them:

```
f x y
    | g x y < 5   = "Less than 5"
    | g x y == 5  = "Equal to 5"
    | otherwise   = "Greater than 5"
    where g a b = 2*a + 3*b
```

[4]Not actually true: $0 \cdot \infty$ is undefined, and $0 \cdot (-1)$ is "negative zero", which is different from "positive zero" in certain programming contexts.

There are more examples of “local variables”, even in the interactive prompt:

```
ghci> let x = 2; y = 3
ghci> let f x y = (x,y)
ghci> f x y
(2,3)
ghci> f 4 5
(4,5)
ghci> let x = 4; y = 5 in f x y
(4,5)
ghci> x
2
ghci> y
3
ghci> let x = 100; y = 200
ghci> f x y
(100,200)
```

What’s going on here? As we know, in Haskell, no “variables” can change.

1. `let x = 2; y = 3` defines the names `x` and `y` to be `2` and `3`.
2. `let f x y = (x,y)` defines a two-parameter function that pairs the parameters (essentially, `(,)`).
3. `f x y` calls `f` with the parameters `x` and `y`, which are `2` and `3`.
4. `f 4 5` calls `f` with `4` and `5`. Because the `x` and `y` from `(x,y)` are logically different than those from `x = 2` and `y = 3`, the function behaves as expected.
5. `let x = 4; y = 5 in f x y` temporarily binds `4` and `5` to `x` and `y` respectively, then calls the function with those values.
6. `x` and `y` have not changed outside the previous expression — they are still `2` and `3`.
7. `let x = 100; y = 200` binds the “new” values of `100` and `200` to the names `x` and `y`
8. `f x y` proves that the new values remain.

What happens is that when we call `let` in 2 and 7, we don’t permanently change their values — if we exit GHCi and enter it again, the defined values are gone.

What the `let` in 2 and 7 does is temporarily bind the values (`2` and `3` and then `100` and `200`) to `x` and `y` until the end of the interactive session.

The `let` in 5 temporarily binds `4` and `5` to `x` and `y` until `f x y` is evaluated, after which it “reverts” to the previous values.

What’s going on here may be confusing, but hopefully it is somewhat intuitive. The point is that we’re not talking of the same `x` and `y` with different values, we’re talking about different `x`s and `y`s. We can illustrate that:

```
ghci> let x1 = 2; y1 = 3
ghci> let f x2 y2 = (x2,y2)
ghci> f x1 y1
(2,3)
ghci> f 4 5
(4,5)
ghci> let x3 = 4; y3 = 5 in f x3 y3
(4,5)
ghci> x1
2
ghci> y1
```

```
3
ghci> let x4 = 100; y4 = 200
ghci> f x4 y4
(100,200)
```

We’re going to get better at understanding the when and how of local “variables” as our experience increases.

# B. Types and Typeclasses

## B.1. Typeclasses in Depth

Typeclasses are the bread and butter of Haskell[1]. Some of the most common (and useful) typeclasses, roughly presented from general to specific, are:

### B.1.1. Show and Read

These two typeclasses are, for the most part, invisible to the user. Although almost every type out there belongs to both of them, `Show` and `Read` are handled by the computer[2] — we only need to tell the compiler "hey, that type is part of `Show`" and it does the rest.

- `Show` contains all types which can be converted to strings.
  - *Includes:* almost all types (`Int`, `[Bool]`, `[[Char]]` etc.)
  - *Does not include:* functions (`Int -> Int` etc.)
  - *Prerequisites:* none
  - *Built-in functions:*
    - `show` converts a value to a string

```
ghci> show 5
"5"
ghci> show 203
"203"
ghci> show False
"False"
ghci> show [1, 2, 5]
"[1,2,5]"
ghci> show ["hi", "hello", "blah"] -- result looks funky
"[\"hi\",\"hello\",\"blah\"]"
```

- `Read` is the converse of `Show`.
  - *Includes:* almost everything that can also be `show`n.
  - *Does not include:* functions
  - *Prerequisites:* none
  - *Built-in functions:*
    - `read` converts a string to a specific value[3].

[1] Author's note: in retrospect, I don't know what I meant by saying this.
[2] They can also, however, be manually specified, but that's rare. The computer does a really good job.
[3] The type has to be specified, either by performing an operation and letting Haskell infer, or by explicitly declaring it. Otherwise, an ambiguous type variable error is thrown (details in B.2.2).

```
ghci> read "True" && False
False
ghci> read "True" :: Bool
True
ghci> read "67" + 89
156
ghci> read "67" -- ambiguous type variable error
```

### B.1.2. Eq, Ord, Enum

Many useful functions require membership in at least one of these typeclasses. After all, there is no function that can order unsortable items, and you can't list that which cannot be enumerated.

- `Eq` contains all types that can be equated.
  - *Includes:* almost all types
  - *Does not include:* functions
  - *Prerequisites:* none
  - *Built-in functions:*
    - `==` tests for equality
    - `/=` tests for inequality

```
ghci> 5 == 6
False
ghci> "hello" == "hello"
True
ghci> (+) == (*) -- type error
```

- `Ord` contains types which have a logical ordering.
  - *Includes:* almost all types
  - *Does not include:* functions
  - *Prerequisites:* `Eq`
  - *Built-in functions:*
    - `>` and `>=`
    - `<` and `<=`
    - `compare` returns an ordering
    - `max` and `min`

```
ghci> 4 > 5
False
ghci> "abcd" >= "abcc"
True
ghci> True < False
False
ghci> max 10 3
10
ghci> compare 4 5
```

```
LT
ghci> compare 4 4
EQ
ghci> compare 5 4
GT
```

- `Enum` contains types which can be enumerated.
  - *Includes:* almost all types
  - *Does not include:* functions, strings
  - *Prerequisites:* `Ord`
  - *Built-in functions:*
    - `succ` returns the logical successor
    - `pred` returns the logical predecessor
    - Other functions synonymous to using ranges

```
ghci> succ 6
7
ghci> succ 'y'
'z'
ghci> succ 'z'
'{'
ghci> succ "abcde" -- type error
```

### B.1.3. Numeric Typeclasses

All numbers have a common set of operations. They can, for example, be added or subtracted, even multiplied. There are grouped in many different classes, however, because some of them lack specific behavior. For instance, complex numbers[4] cannot be ordered[5].

- `Num` is the most general numeric typeclass.
  - *Includes:* `Int`, `Integer`, `Rational`, `Float`, `Double` etc.
  - *Does not include:* non-numbers
  - *Prerequisites:* `Eq`, `Show`
  - *Built-in functions:*
    - `+`, `-`, and `*`
    - `negate` returns the opposite of a number
    - `abs` returns the absolute value
    - `signum` is the sign function[6]

```
ghci> 5 + 4 * 3 - 2
15
ghci> negate 10
```

[4]The issue is multifaceted: complex numbers have the type `RealFloat a => Complex a` as opposed to other numbers (for example, `Fractional a => a`).

[5]The previous footnote was about complex numbers, not their ordering. Just a clarification.

[6]Returns `1` on a positive number, `0` on zero, and `-1` on a negative number.

```
-10
ghci> abs (-5)
5
ghci> signum 23
1
ghci> signum (-23)
-1
```

- `Integral` is the typeclass of integers.
  - *Includes:* `Int`, `Integer` and other size integers (`Int8`, `Int16`, `Int32` etc.)
  - *Does not include:* anything else
  - *Prerequisites:* `Num`, `Ord`, `Enum`
  - *Built-in functions:*
    - `quot`, the quontient in division
    - `div`, integer division
    - `rem`, the remainder
    - `mod`, modulo function

```
ghci> 17 `quot` 3
5
ghci> 17 `div` 3
5
ghci> 17 `rem` 3
2
ghci> 17 `mod` 3
2
ghci> 17 `quot` (-3)
-5
ghci> 17 `div` (-3)
-6
ghci> 17 `rem` (-3)
2
ghci> 17 `mod` (-3)
-1
```

***Warning!*** Do not confuse `quot` with `div` and `rem` with `mod` — they behave differently on negatives.

- `Fractional` contains fractions, both common ($\frac{1}{4}$, $\frac{2}{3}$) and decimal (2.5, 8.53)
  - *Includes:* `Rational`, `Float`, `Double` etc.
  - *Does not include:* integers, non-numbers
  - *Prerequisites:* `Num`
  - *Built-in functions:*
    - `/`, the division function
    - `recip`, the inverse of a number ($\frac{1}{x}$, where $x$ is the number)[7]

[7] `recip 0` gives `Infinity`. However, `Infinity` is not a number *per se*, it's just a way to display $\infty$. If we really want to use $\infty$ in our calculations (which, by the way, is an extremely bad idea), we must use `recip 0` or `1/0` or whatever.

```
ghci> 5.2 / 3.2
1.625
ghci> recip 0.25
4.0
ghci> 1 / 0.25
4.0
ghci> recip 0
Infinity
ghci> 1 / Infinity -- doesn't work
```

- `Floating` contains decimal numbers
  - *Includes:* `Float`, `Double` etc.
  - *Does not include:* common fractions, integers, non-numbers
  - *Prerequisites:* `Fractional`
  - *Built-in functions:*
    - `pi`, a function of zero parameters (a constant)[8]
    - `exp`, `sqrt`, `log`
    - `logBase`, which takes two parameters
    - `**`, the fractional power function
    - `sin`, `cos`, `tan` and friends (`sinh`, `acos`, `asinh` etc.)

```
ghci> pi :: Float
3.1415927
ghci> pi :: Double
3.141592653589793
ghci> log 10
2.302585092994046
ghci> 5 ** 2.3
40.51641491731905
ghci> sin (pi / 3)
0.8660254037844386
ghci> cos (pi / 3)
0.5000000000000001
ghci> logBase 10 1000
2.9999999999999996
```

***Warning!*** Watch out for rounding errors — they're a pain in the brain.

## B.2. Type Errors

### B.2.1. General Type Errors

We'll analyze the following type error in detail, line by line. Intimate knowledge of the structure of type errors should help us fix them much faster.

[8]It's a very interesting case — because functions can be polymorphic and constants are (zero-parameter) functions, constants can also be polymorphic.

```
ghci> 1 * False

<interactive>:1:1:
    No instance for (Num Bool)
      arising from the literal ‘1’
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘(*)’, namely ‘1’
    In the expression: 1 * False
    In an equation for ‘it’: it = 1 * False
```

The analysis, as promised:

1. `ghci> 1 * False` is the (incorrect) expression we ran.
2. is a blank line. It doesn't really do anything.
3. `<interactive>:1:1:` is the location in the program that gives the error (`[line]:[character]`).
4. `No instance for (Num Bool)` means that `False`, which is a `Bool`, can't be a number (`Num`).
5. `arising from the literal ‘1’` tells us that it is through our use of `1`, which is a number, GHCi inferred that `False` must also be a number so it can multiply them. But `False` is a `Bool`, and `Bool`s aren't numbers. *Contradiction.*
6. `Possible fix: add an instance declaration for (Num Bool)` suggests that it is possible to fix the error by defining how `Bool`s can be numbers. For example, if we tell GHCi that `False` is the same as `0` and `True` is really `1`, then the expression would compile[9]. Adding instance declarations is explained in [XREF].
7. `In the first argument of ‘(*)’, namely ‘1’` gives specific context for the error: the first argument of `*`.
8. `In the expression: 1 * False` gives more general context.
9. `In an equation for ‘it’: it = 1 * False` gives the most general context of the error. In GHCi, `it` is an internal variable that stores the result of the previous computation.

Basically all type errors in GHCi follow the above format[10]. It's important to understand them as they're the fastest way of identifying the problem, especially in very complex cases.

### B.2.2. Ambiguous Type Variable Errors

Sometimes Haskell cannot successfully infer the types of the expressions involved. In that case, we are presented with the following:

```
ghci> read "5"

<interactive>:1:1:
    Ambiguous type variable ‘a0’ in the constraint:
      (Read a0) arising from a use of ‘read’
    Probable fix: add a type signature that fixes these type variable(s)
    In the expression: read "5"
    In an equation for ‘it’: it = read "5"
```

We shall, yet again, dissect the error. The line-by-line analysis shows that:

1. `ghci> read` “5” is our ambiguous expression.

[9] In this case it's not recommended, seeing how multiplying a `Bool` and a number doesn't make much sense.
[10] Other interpreters may display differently.

2. is an empty line. GHCi has the tendency to put that before long errors.
3. `<interactive>:1:1:` is the position of the ambiguous statement, `[line]:[character]`. Here, it's at the very beginning of our interactive statement.
4. `Ambiguous type variable ‘a0’ in the constraint:` tells us that GHCi cannot infer the type because it has multiple "solutions".
5. `(Read a0) arising from a use of ‘read’` indicates that the typeclass `Read` contains multiple types. What it doesn't say, but we know, is that Haskell must know the specific type to `read`. For example, `5` can be `read` as:
   a) A character (`'5'`)
   b) A number (`5`)
   c) A string (`"5"`)
   d) Many, many others
6. `Probable fix: add a type signature that fixes these type variable(s)` recommends fixing the error by adding an expicit type signature[11]. GHCi implies ("probable") that in most cases this would be the desireable action.
7. `In the expression: read "5"` is the context of the ambiguity.
8. `In an equation for ‘it’: it = read "5"` gives even more context. With all this info, it's hard not to identify and fix the problem immediately!

### B.2.3. Making Custom Errors

A more expressive way of "correcting" a program without actually suppressing the error is to write our own error message. We might want this if it's the user's fault for incorrect input, and we want to halt the program, as well as help the user in fixing the input. We will use the 4.1.1 base example, reproduced below for convenience.

```haskell
-- File: patterns-wrong.hs

intToString :: Int -> [Char]
intToString 1 = "one"
intToString 2 = "two"
intToString 3 = "three"
```

The `error` function takes a string and throws an error with that message.

```haskell
-- File: patterns-wrong.hs (FIXED)

intToString :: Int -> [Char]
intToString 1 = "one"
intToString 2 = "two"
intToString 3 = "three"
intToString _ = error "intToString: Number too large"
```

Notice that the error handler doesn't know the name of the function beforehand, so we might want to include it in the error message, like above.

```
ghci> intToString 20
*** Exception: intToString: Number too large
```

[11]Such as `:: Int` or `:: Char`.

Because `error` is an ordinary function, we can also do some magic to make it more expressive.

```haskell
-- File: patterns-wrong.hs (FIXED)

intToString :: Int -> [Char]
intToString 1 = "one"
intToString 2 = "two"
intToString 3 = "three"
intToString n = error ("intToString: Number " ++ show n ++ " too large")
```

```
ghci> intToString 20
*** Exception: intToString: Number 20 too large
```

Cusomizing error messages is not mandatory, but it's a very good idea, especially in long and complicated programs. Of course, the real solution is never to crash expressively, but to actually aid the user without blowing the program to smithereens: *graceful failure*. We learn such methods late in the book, in [XREF] and [XREF].

# C. Modules

## C.1. Data.List

The `Data.List` module is the one-stop shop for all our list goodies. It supports many functions, detailed below. The trickier ones have example code.

[FIXME]

# D. Hints to Exercises

## D.1. Introduction

### D.1.1. About the Book

This would be the place that the hints to this exercise will be located. Watch out not to accidentally spoil adjacent exercises when you read! I'd personally suggest reading one sentence at a time with pauses in between.

1. This would be a hint to the first exercise.
    a) Any additional hints
    b) would look like this.

### D.1.2. Why Haskell?

Since we're not really into the book yet, I figure I'd give my own answers to these open questions here. Practice not reading ahead here.

1. I guess I first learned Haskell because I wanted to try out programming languages and see if I had a favorite.
    a) I immediately fell in love with Haskell.
    b) This was consolidated when I tried XMonad out...
    c) And here I am! I was sure I'd get bored of it in a couple of weeks but it stuck for some reason.
2. I actually knew little programming when I started out, maybe a bit of C.
    a) Actually, I don't know much programming now either! I like it, but not enough to do it on a constant basis.[1]
    b) I usually program just for the hell of it and can't really see myself doing this as my primary job.[2]
    c) But yeah, back when I started with Haskell, I think not really knowing any programming helped me.
    d) I was more open-minded about things (because I didn't know much of anything!) and didn't really find it weird that there are no variables, or the strange meaning that Haskell gives to "classes".
3. *This is not the third answer.
4. **This is the answer to the fourth exercise. This one would be obscenely detailed because the problem is really hard!

[1]It's been almost two years since I wrote this sentence. I really enjoy programming! I'd do it all the time!
[2]How times have changed! I could totally see myself doing programming as my primary job!

### D.1.3. Before We Start

1. I honestly can't help you with this one. It depends on your choice and what operating system you run. Check the following things:
   a) How does the manual/website instruct you to open the interpreter?
   b) Do you need to have any additional programs installed?
   c) Is there any specific error message that is given? Try to look it up on Google.
2. If it won't load, what does the error tell you?
   a) Is `starting-out.hs` in a different directory than the one you opened GHCi (or other interpreter) in? If so, what do you need to do?
   b) You don't have to move `starting-out.hs` to that directory (but it works).
   c) Does it say anything about syntax? You might want to check if you made any typos.
   d) Take a look at the example file and interactive session. They should look similar.
3. How do you test for equality?
   a) The test for equality is not `=`.
   b) However, it is very similar. Do not confuse `=` with the real one.
   c) Take a look at the example interactive session. We've done something similar there.
4. Have you tried adding a new line at the end that says `b = 5`?
   a) That won't work. You can't usually define a variable twice.
   b) You're stuck with changing the initial definition of `b`. Don't worry, in the real world you almost never need to define something twice!

## D.2. Basics: Functions and Lists

### D.2.1. Getting Started

1. The answer should be `2.551somethingsomething`. I think.
   a) The idea of this exercise was to emphasize the importance of readability in your code. If the code isn't readable, all sorts of problems will inevitably occur: the code will be hard to modify, hard to maintain, hard to understand. Errors will creep in and ruin hours of work. It's a disaster.
   b) Some of the techniques to improve readability are (more on these later):
      i. Commenting the code
      ii. Splitting large functions into smaller ones
      iii. Using some of the fancier features of Haskell
      iv. Making functions more abstract
2. `max` only accepts two parameters, but you need three. What do you do?
   a) One of the solutions uses `max` twice and also needs a pair of parentheses.
   b) What is `max 2 3`? Can you feed it further along the line?
   c) One other solution involves no parentheses at all and it's intuitive.

d) What would you do if you needed to add three numbers? What feature of the language enables `max` to work the same way?

3. We've discussed `max` already. Addition and multiplication are pretty straightforward. The real problem might be with equality.

   a) Did you do `x == y == z`? Why doesn't it work?

   b) Equality is not associative. To fix that, have you tried grouping the expressions with parentheses?

   c) It won't work. Why not?

   d) Remember the functions that operate on booleans? You need one of them. Which one?

4. *One of the solutions uses a purely mathematical method that only works with numbers. The other one uses a feature of the language that we've only touched once so far.

   a) How would you calculate the maximum between two numbers using only mathematical operators?

   b) Remember that only `max` and `min` are disallowed. You might want to use a special function that we've only mentioned by name.

   c) It's the modulus function that gives the absolute value of a number. What is its name in Haskell?

   d) The function you want is `abs`. You must combine `+`, `-`, `abs`, and `/` in a coherent function that does what you want.

   e) The other solution works on everything that can be ordered, not just numbers. It requires a feature of the language that lets us do different things depending if a condition is true or not.

   f) What does `max x y` return if `x = 2` and `y = 3`, `x` or `y`? What if `x = 3` and `y = 2`?

   g) The condition that you want to test for is if `x < y`. What do you do now? Look back on what we've seen so far. It's in one of the code examples.

   h) Again, this exercise has a purpose that is beyond academic. One of the solutions is narrow (only works on numbers) and not terribly elegant[3]. The other works on all possible inputs with one simple condition (that they can be ordered), is more efficient and much easier to read. The second solution is obviously preferable but in Haskell such solutions are seldom obvious or trivial. We'll need to work for them.

[3]Not to mention that giving it two integers returns a fractional number which might make other functions down the line (those that expect to be given integers) choke on them.