Contents

Haskell for CMI

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

This is (still!) an incomplete draft.
Please send any corrections, comments etc. to feedback_host@mailthing.com
Last updated June 08, 2025.

To someone

Table of Contents

Basic Theory

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

Mathematics vs Haskell (taking suggestions for a better heading)

The main difference between mathematics and haskell is **who** reads what we write.

When writing any form of mathematical expression, it is the expectation that it is meant to be read by humans, and convince them of some mathematical proposition.
On the other hand, haskell code is not *primarily* meant to be read by humans, but rather by machines. The computer reads haskell code, and tries to interpret it into steps of manipulating expressions.

When writing mathematics, we can choose to be a bit sloppy and hand-wavy with our words, as we can rely to some degree on the imagination and pattern-sensing abilities of the reader to fill in the gaps.

However, in this context, computers, being unintelligent machines, are extremely dumb and stupid. Unless we spell out the details for them in excruciating detail, they are not going to understand what we want them to do.

Since in this course we are going to be writing for computers, we need to ensure that our writing is very precise, correct and generally **idiot-proof**. (Because, in short, computers are idiots)

In order to practice this more formal style of writing required for **haskell code**, the first step we can take is to know how to write our familiar **mathematics** more formally.

The Building Blocks

The language of writing mathematics is fundamentally based on two things -

- **Symbols:** such as $0, 1, 2, 3, x, y, z, n, \alpha, \gamma, \delta, \mathbb{N}, \mathbb{Q}, \mathbb{R}, \in, <, >, f, g, h, \Rightarrow, \forall, \exists$ etc.

and

- **Expressions:** which are sentences or phrases made by chaining together these symbols, such as
  - $x^3 \cdot x^5 + x^2 + 1$
  - $f(g(x, y), f(a, h(v), c), h(h(h(n))))$
  - $\forall \alpha \in \mathbb{R}\ \exists L \in \mathbb{R}\ \forall \varepsilon > 0\ \exists \delta > 0\ \ \mid x - \alpha \mid < \delta \Rightarrow \mid f(x) - f(\alpha) \mid < \varepsilon$

  etc.

Values

mathematical value

A mathematical **value** is a single and specific well-defined mathematical object that is constant, i.e., does not change from scenario to scenario nor represents an arbitrary object.

Examples include -

- The real number π
- The order $<$ on $\mathbb{N}$
- The function of squaring a real number : $\mathbb{R} \to \mathbb{R}$
- The number of non-trivial zeroes of the Riemann Zeta function

Therefore we can see that relations and functions can also be **values**, as long as they are constant, specific, and not scenario-dependent.

In fact, as we see in the last example, even if we don't know what the exact value is, we can still know that it is **some value**,
as it is a constant, even though it is an unknown constant.

Variables

mathematical variable

A mathematical **variable** is a symbol or chain of symbols meant to represent a value that is arbitrary in some way, usually as a way to show that whatever process follows can be carried out with any arbitrary value.

For example, consider the following theorem -

Theorem Adding 1 to a natural number makes it bigger.

Proof Take n to be an arbitrary natural number.
We know that $1 > 0$.
Adding n to both sides of the preceding inequality yields

$$n + 1 > n$$ ■

Here, n is a variable as it isn't any specific value, but rather an arbitrary instance of a certain type of value.
It has been used to show a certain fact that holds for **any** natural number.

Well-Formed Expressions

well-formed mathematical expression

It is difficult to give a direct definition of a **well-formed expression**. As an alternative, we can define a ***formal procedure*** to check whether an expression is well-formed or not.

The procedure is as follows -

Given an expression e,

- first check whether e is a
  - ≒ **mathematical value**
  or
  - ≒ **mathematical variable**
  in which cases e passes the check and is an expression.

Failing that,

- check whether e is of the form $f(e_1, e_2, e_3, ..., e_n)$, where
  - f is a function (the function can be a ≒ **mathematical value** or ≒ **mathematical variable**)
  - which takes n inputs,
  and
  - $e_1, e_2, e_3, ..., e_n$ are all ***well-formed expressions*** which are ***valid inputs*** to f.

Let us use this defining procedure to check if $x^3 \cdot x^5 + x^2 + 1$ is a well-formed expression.
(We will skip the check of whether something is a valid input or not, as that notion is still not very well-defined for us.)

$x^3 \cdot x^5 + x^2 + 1$ is $+$ applied to the inputs $x^3 \cdot x^5$ and $x^2 + 1$.
Thus we need to check that $x^3 \cdot x^5$ and $x^2 + 1$ are well-formed expressions which are valid inputs to $+$.

$x^3 \cdot x^5$ is $\cdot$ applied to the inputs x^3 and x^5.
Thus we need to check that x^3 and x^5 are well-formed expressions.

x^3 is $(\ )^3$ applied to the input x.
Thus we need to check that x is a well-formed expression.

x is a well-formed expression, as it is a ÷ **mathematical variable**.

x^5 is $(\ )^5$ applied to the input x.
Thus we need to check that x is a well-formed expression.

x is a well-formed expression, as it is a ÷ **mathematical variable**.

$x^2 + 1$ is $+$ applied to the inputs x^2 and 1.
Thus we need to check that x^2 and 1 are well-formed expressions.

x^2 is $(\ )^2$ applied to the input x.
Thus we need to check that x is a well-formed expression.

x is a well-formed expression, as it is a ÷ **mathematical variable**.

1 is a well-formed expression, as it is a ÷ **mathematical value**.

Done!

x **checking well-formedness of mathematical expression**

Check whether $f(g(x, y), f(a, h(v), c), h(h(h(n))))$ is a well-formed expression or not.

Defining Functions

Functions are a very important tool in mathematics and they form the foundations of Haskell programming.
Thus, it is very helpful to have a deeper understanding of how they are defined.

Using Expressions

In its simplest form, a definition of a function is made up of a left-hand side, '$:=$' in the middle[1], and a right-hand side.

On the left we write the name of the function followed by a number of variables which represent its inputs.

In the middle we write '$:=$', indicating that right-hand side is the definition of the left-hand side.

On the right, we write a ÷ **well-formed mathematical expression** using the variables of the left-hand side, describing to how to combine and manipulate the inputs to form the output of the function.

A few examples -
- $f(x) := x^3 \cdot x^5 + x^2 + 1$
- more examples

Some Conveniences

Often in the complicated definitions of some functions, the right-hand side expression can get very convoluted, so there are some conveniences which we can use to reduce this mess.

[1] In order to have a clear distinction between notation and equality,
we use $A := B$ to mean "A is defined to be B",
and we use $A == B$ to mean "A is equal to B".

Where, Let

which, for convenience, can be rewritten as -

or as -

Anonymous Functions

which, for convenience, can be rewritten as -

which is particularly useful when we (for some reason) do not want name the function.

This notation can also be used when there are multiple inputs.

Consider -

which, for convenience, can be rewritten as -

Piecewise Functions

Pattern Matching

which, for convenience, can be rewritten as -

Recursion

A function definition is recursive when the name of the function being defined appears on the right-hand side as well.

For example, consider defining the famous fibonacci function -

$$\begin{aligned} F &: \mathbb{N} \to \mathbb{N} \\ F(0) &:= 1 \\ F(1) &:= 1 \\ F(n) &:= F(n-1) + F(n-2) \end{aligned}$$

Termination

But it might happen that a recursive definition might not give a final output for a certain input.

For example, consider the following definition -

$$f(n) := f(n+1)$$

It is obvious that this definition does not define an actual output for, say, $f(4)$.

However, the previous definition of F obviously defines a specific output for $F(4)$ as follows -

$$
\begin{aligned}
F(4) &== F(3) + F(2) \\
&== (F(2) + F(1)) + F(2) \\
&== ((F(1) + F(0)) + F(1)) + F(2) \\
&== ((1 + F(0)) + F(1)) + F(2) \\
&== ((1 + 1) + F(1)) + F(2) \\
&== (2 + F(1)) + F(2) \\
&== (2 + 1) + F(2) \\
&== 3 + F(2) \\
&== 3 + (F(1) + F(0)) \\
&== 3 + (1 + F(0)) \\
&== 3 + (1 + 1) \\
&== 3 + 2 \\
&== 5
\end{aligned}
$$

÷ termination of recursive definition

In general, a recursive definition is said to **terminate on an input**
$\Leftrightarrow$ it eventually gives an ***actual specific output for that input***.

But what we cannot do this for every $F(n)$ one by one.

What we can do instead, is use a powerful tool known as the **÷ principle of mathematical induction**.

Induction

÷ principle of mathematical induction

If we have an infinite sequence of statements $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \dots$
and we can prove the following 2 statements -

- φ_0
- $\forall n(\varphi_n \Rightarrow \varphi_{n+1})$

then all the statements $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \dots$ in the sequence are true.

Proving Termination using Induction

So let's see the **÷ principle of mathematical induction** in action, and use it to prove that

Theorem The definition of the fibonacci function F terminates for any natural number n.

Proof For each natural number n, let φ_n be the statement

" The definition of F terminates for every natural number which is $\leq n$ "

To apply the **÷ principle of mathematical induction**, we need only prove the 2 requirements and we'll be done. So let's do that -

- $\langle\langle\ \varphi_0\ \rangle\rangle$
  The only natural number which is ≤ 0 is 0, and $F(0) := 1$, so the definition terminates immediately.

- $\langle\langle\ \forall n(\varphi_n \Rightarrow \varphi_{n+1})\ \rangle\rangle$
  Assume that φ_n is true.
  Let m be an arbitrary natural number which is $\leq n+1$.
  - $\langle\langle$ Case 1 $(m \leq 1)$ $\rangle\rangle$
    $F(m) := 1$, so the definition terminates immediately.
  - $\langle\langle$ Case 2 $(m > 1)$ $\rangle\rangle$
    $F(m) := F(m-1) + F(m-2)$,
    and since $m-1$ and $m-2$ are both $\leq n$,
    φ_n tells us that both $F(m-1)$ and $F(m-2)$ must terminate.
    Thus $F(m) := F(m-1) + F(m-2)$ must also terminate.

  Hence φ_{n+1} is proved! ■

Trees

Trees are a way to meaningfully structure a collection of objects. Understanding the meaning captured by these structures is vitally important in learning about expressions.

In fact, the internal structure of any object in Haskell is modelled as a tree-like structure.

Definition

We will adopt a similar approach to defining trees as we did with expressions, i.e., we will provide a formal procedure to check whether a mathematical object is a tree, rather than directly defining what a tree is.

≒ **tree**

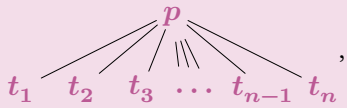
A **tree over a set S** defines a meaningful structure on a collection of elements of S.

The procedure to determine whether an object is a **tree over a set S** is as follows -

Given a mathematical object t,

- first check whether $t \in S$, in which case t passes the check, and is a **tree over S**

Failing that,

- check whether t is of the form 

, where
  - $p \in S$
  - and each of $t_1, t_2, t_3, ..., t_{n-1}$, and t_n is a **tree over S**.

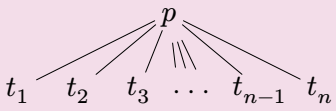
Structural Induction

In order to prove things about trees, we have a version of the ≒ **principle of mathematical induction** for trees -

structural induction for trees

If for each tree t over a set S, we have a statement φ_t,

and we can prove the following two statements -

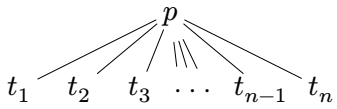
- $\forall s \in S, \varphi_s$ is true
- $\forall p \in S$,
  $\forall$ trees $t_1, t_2, t_3, ..., t_{n-1}, t_n$ over S,
  $\left(\varphi_{t_1} \text{ and } \varphi_{t_2} \text{ and } \varphi_{t_3} \text{ and ... and } \varphi_{t_{n-1}} \text{ and } \varphi_{t_n}\right) \Rightarrow \varphi_\tau$,
  where $\tau ==$ 

then φ_t is true for each tree t over S.

Structural Recursion

We can also define functions on trees using a certain style of recursion.

From the definition of **tree**, we know that trees are

- either of the form $s \in S$
- or of the form 

So, to define any function (f : Trees over $S \to X$), we can divide taking the input into two cases, and define the outputs respectively.

Let's use this principle to define the function

$$\text{size} : \text{Trees over } S \to \mathbb{N}$$

which is meant to give the number of elements of S which appear in a tree over S.

$\text{size}(s) := 1$

$$\text{size}\left(\begin{array}{c} p \\ t_1 \quad t_2 \quad t_3 \quad \cdots \quad t_{n-1} \quad t_n \end{array} \right) := 1 + \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + ... + \text{size}(t_{n-1}) + \text{size}(t_n)$$

Termination

Using **structural induction for trees**, let us prove that

Theorem The definition of the function size terminates on any finite tree.

Proof For each tree t, let φ_t be the statement

" The definition of size terminates on t"

To apply **structural induction for trees**, we need only prove the 2 requirements and we'll be done. So let's do that -

- $\langle\langle\ \forall s \in S, \varphi_s \text{ is true}\ \rangle\rangle$
  $\text{size}(s) := 1$, so the definition terminates immediately.

- $\langle\langle\ \forall p \in S, ..., \left(\varphi_{t_1}\ \textbf{and}\ \varphi_{t_2}\ \textbf{and}\ \varphi_{t_3}\ \textbf{and ... and}\ \varphi_{t_{n-1}}\ \textbf{and}\ \varphi_{t_n}\right) \Rightarrow \varphi_\tau\ \rangle\rangle$
  Assume that each of $\varphi_{t_1},\ \varphi_{t_2},\ \varphi_{t_3}, ...,\ \varphi_{t_{n-1}},\ \varphi_{t_n}$ is true.
  That means that each of $\text{size}(t_1),\ \text{size}(t_2),\ \text{size}(t_3), ...,\ \text{size}(t_{n-1}),\ \text{size}(t_n)$ will terminate.

  Now, $\text{size}(\tau) := 1 + \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + ... + \text{size}(t_{n-1}) + \text{size}(t_n)$

  Thus, we can see that each term in the right-hand side terminates.
  Therefore, the left-hand side "$\text{size}(\tau)$",
  being defined as a well-defined combination of these terms,
  must also terminate.

  Hence φ_τ is proved! ■

Why Trees?

But why care so much about trees anyway? Well, that is mainly due to the previously mentioned fact - "**In fact, the internal structure of any object in Haskell is modelled as a tree-like structure.**"

But why would Haskell choose to do that? There is a good reason, as we are going to see.

The Problem

The Solution

Haskell Setup on Linux

Shubh Sharma

setup linux (feel free to change it)

Haskell Setup on MacOS

Arjun Maneesh Agarwal

setup mac (feel free to change it)

Contents

Haskell Setup on Windows

Ryan Hota

setup win (feel free to change it)

Basic Syntax

Arjun Maneesh Agarwal

Bool, Int, Integer and more (feel free to change it)

Introduction to Types

Haskell is a strictly typed language. This means, Haskell needs to strictly know what the type of **anything** and everything is.

But one would ask here, what is type? According to Cambridge dictionary,

> **Type** refers to a particular group of things that share similar characteristics and form a smaller division of a larger set

Haskell being strict implies that it needs to know the type of everything it deals with. For example,

- The type of e is `Real`.
- The type of 2 is `Int`, for integer.
- The type of 2 can also be `Real`. But the `2 :: Int` and `2 :: Real` are different, because they have different types.
- The type of $x \mapsto \lfloor x \rfloor$ is `Real → Int`, because it takes a real number to an integer.
- We write $(x \mapsto \lfloor x \rfloor)e = 2$ By applying a function of type `Real → Int` to something of type `Real` we get something of type `Int`
- The type of $x \mapsto x + 2$, when it takes integers, is `Int → Int`.
- We cannot write $(x \mapsto x + 2)(e)$, because the types don't match. The function wants an input of type `Int` but e is of type `Real`. We could define a new function $x \mapsto x + 2$ of type `Real → Real`, but it is a different function.
- Functions can return functions. Think of $(+)$ as a function that takes an `Int`, like 3, and returns a function like $x \mapsto x + 3$, which has type `Int → Int` Concretely, $(+)$ is $x \mapsto (y \mapsto y + x)$. This has type `Int → (Int → Int)`.
- We write $(+)(3)(4) = 7$. First, $(+)$ has type `Int → (Int → Int)`, so $(+)(3)$ has type `Int → Int`. So, $(+)(3)(4)$ should have type `Int`.
- The type of $x \mapsto 2 * x$ is `Int → Int` when it takes integers to integers. It can also be `Real → Real` when it takes reals to reals. These are two different functions, because they have different types. But if we make a 'super type' or **typeclass** called `Num` is which is a property which both `Int` and `Real` have, then we can define $x \mapsto 2 * x$ more generally as of type `Num a ⇒ a → a` which reads, for a type `a` with property(belonging to) `Num`, the function $x \mapsto 2 * x$ has type `a → a`
- Similarly, one could define a generalized version of the other functions we described.

A study of types and what we can infer from them(and how we can infer them) is called, rightfully so, **Type Theory**. It is deeply related to computational proof checking and formal verification. While we will not study about it in too much detail in this course, it is its own subject and is covered in detail in other courses.

While we recommend, atleast for the early chapters, to declare the types of your functions explicitly ex. `(+) :: Int → Int → Int`; Haskell has a type inference system[2] which is quite accurate and

[2] Damas–Hindley–Milner Type Inference is the one used in Haskell at time of writing.

tries to go for the most general type. This can be both a blessing and curse, as we will see in a few moments.

This chapter will deal (in varying amounts of details) with the types `Bool`, `Int`, `Integer`, `Float`, `Char` and `String`.

> `Bool` is a type which has only two valid values, `True` and `False`. It most commonly used as output for indicator functions(indicate if something is true or not).

> `Int` and `Integer` are the types used to represent integers.
>
> `Integer` can hold any number no matter how big, up to the limit of your machine's memory, while `Int` corresponds to the set of positive and negative integers that can be expressed in 32 or 64 bits(based on system) with the bounds changing depending on implementation (guaranteed at least -2^29 to 2^29). Going outside this range may give weird results. Ex. `product [1..52] :: Int` gives a negative number which cannot realistically be 52!. On the other hand, `product [1..52] :: Integer` gives indeed the correct answer.
>
> The reason for `Int` existing despite its bounds and us not using `Integer` for everything is related to speed and memory. Using the former is faster and uses lesser memory.
>
> ```
> >>> product [1..52] :: Int
> -8452693550620999680
> (0.02 secs, 87,896 bytes)
> >>> product [1..52] :: Integer
> 80658175170943878571660636856403766975289505440883277824000000000000
> (0.02 secs, 123,256 bytes)
> ```
>
> Almost 1.5 times more memory is used in this case.

An irrefutable fact is that computers are fundamentally limited by the amount of data they can keep and humans are fundamentality limited by the amount of time they have. This implies that if, we can optimize for speed and space, we should do so. We will talk some more about this in [chapter 9], but the rule of thumb is that more we know about the input, the more we can optimize. Knowing that it will be between, say -2^{29} to 2^{29}, allows for some optimizations which can't be done with arbitrary length. We (may) see some of these optimizations later.

`Rational`, `Float` and `Double` are the types used to deal with non-integral numbers. The former is used for fractions or rationals while the latter for reals with varying amount of precision. Rationals are declared using `%` as the viniculum(the dash between numerator and denominator). For example `1%3`, `2%5`, `97%31`.

`Float` or Floating point contains numbers with a decimal point with a fixed amount of memory being used for their storage. The term floating-point comes from the fact that the number of digits permitted after the decimal point depends upon the magnitude of the number. The same can be said for `Double` or Double Precision Floating Point which offers double the space beyond the point, at cost of more memory. For example

```
>>> sqrt 2 :: Float
1.4142135
>>> sqrt 99999 :: Float
316.2262
>>> sqrt 2 :: Double
1.4142135623730951
>>> sqrt 99999 :: Double
316.226184874055
>>> sqrt 999999999 :: Double
31622.776585872405
```

We can see that the prescission of $\sqrt{99999}$ is much lower than that of $\sqrt{2}$. We will use `Float` for most of this book.

`Char` are the types used to represent arbitrary Unicode characters. This includes all numbers, letters, white spaces(space, tab, newline etc) and other special characters.

`String` is the type used to represent a bunch of characters chained together. Every word, sentence, paragraph is either a string or a collection of them.

In haskell, Strings and Chars are differentiated using the type of quotation used. `"hello" :: String` as well as `"H" :: String` but `'H' :: Char`. Unlike some other languages, like say Python, we can't do so interchangeably. Double Quotes for Strings and Single Quotes for Chars.

Similer to many modern languages, In Haskell, String is just a synonym for a list of characters that is `String` is same as `[Char]`. This allows string manipulation to be extremely easy in Haskell and is one of the reason why Pandoc, a universal document converter and one of the most used software in the world, is written in Haskell. We will try to make a mini version of this at the end of the chapter.

To recall, a tuple is a length immutable, ordered multi-typed data structure. This means we can store a fixed number of multiple types of data in an order using tuples. Ex. `(False , True ) :: (Bool, Bool)` `(False , 'a', True ) :: (Bool, Char, Bool)` `("Yes", 5.21 , 'a') :: (String, Float, Char)`

A list is a length mutable, ordered, single typed data structure. This means we can store an arbitrary number things of the same type in a certain order using lists. Ex. `[False, True, False] :: [Bool]` `['a','b','c','d'] :: [Char]` `["One","Two","Three"] :: [String]`

Logical Operations

For example -
Write Haskell code to simulate the following logical operators

1. NOT
2. OR
3. AND
4. NAND
5. XOR

Implementing a not operator seems the most straightforward and it indeed is. We can simply specify the output for all the cases, as there are only 2.

```
not :: Bool → Bool
not True = False
not False = True
```

The inbuilt function is also called `not`. We could employ a smiler strategy for `or` to get the following code

```
or :: Bool → Bool → Bool
or True True = True
or True False = True
or False True = True
or False False = False
```

but this is too verbose. One could write a better code using wildcards as follows

```
or :: Bool → Bool → Bool
or False False = False
or _ _ = True
```

As the first statement is checked against first, the only false case is evaluated and if it is not satisfied, we just return true. We can write this as a one liner using the if statement.

```
or :: Bool → Bool → Bool
or a b = if (a,b) == (False, False) then False else True
```

The inbuilt operator for this is `||` used as `False || True` which evaluates to `True`.

How would one write such a code for `and`? This is left as exercise for the reader. The inbuilt operator for this is `&&` used as `True && False` which evaluates to `False`.

Now that we already have `and` and `not`, could we make `nand` by just composing them? Sure.

```
nand :: Bool → Bool → Bool
nand a b = not (a && b)
```

This also seems like as good of a time as any to introduce operation conversion and function composition. In Haskell, functions are first class citizens. It is a functional programming language after all. Given two functions, we naturally want to compose them. Say we want to make the function $h(x) : x \mapsto -x^2$ and we have $g(x) : x \mapsto x^2$ and $f(x) : x \mapsto -x$. So we can define $h(x) := (f \circ g)(x) = f(g(x))$. In haskell, this would look like

```
negate :: Int → Int
negate x = - x

square :: Int → Int
square x = x^2

negateSquare :: Int → Int
negateSquare x = negate . square
```

We could also define `negateSquare` in a more cumbersome `negateSquare x = negate(square x)` but with complicated expressions these brackets will add up and we want to avoid them as far as possible. We will also now talk about the fact that the infix operators, like `+, -, *, /, ^, &&, ||` etc are also deep inside functions. This means we can should be able to access them as functions(to maybe compose them) as well as make our own. And we indeed can, the method is brackets and backticks.

An operator inside a bracket is a function and a function in backticks is an operator. For example

```
>>> True && False
False
>>> (&&) True False
False
>>> f x y = x*y + x + y
>>> f 3 4
19
>>> 3 `f` 4
19
```

All this means, we could define `nand` simply as

```
nand :: Bool → Bool → Bool
nand = not . (&&)
```

Furthermore, as Haskell doesn't have an inbuilt nand operator, say I want to have `@@` to represent it. Then, I could write

```haskell
(@@) :: Bool → Bool → Bool
(@@) = not.(&&)
```

Finally, we need to make `xor`. We will now replicate a classic example of 17 ways to define it and a quick reference for a lot of the syntax.

λ 17 Xors

```haskell
-- Notice, we can declare the type of a bunch of functions by comma
seperating them.

xor1, xor2, xor3, xor4, xor5 :: Bool → Bool → Bool

-- Explaining the output for each and every case.
xor1 False False = False
xor1 False True = True
xor1 True False = True
xor1 True True = False

-- We could be smarter and save some keystrokes
xor2 False b = b
xor2 b False = b
xor2 b1 b2 = False

-- This seems to to be the same length but notice, b1 and b2 are just
names never used again. This means..
xor3 False True = True
xor3 True False = True
xor3 b1 b2 = False

-- .. we can replace them with wildcards.
xor4 False True = True
xor4 True False = True
xor4 _ _ = False


-- Although, a simple observation recduces work further. Notice, we can't
replace b with a wild card here as it is used in the defination later and
we wish to refer to it.
xor5 False b = b
xor5 True b = not b
```

All the above methods basically enumerate all possibilities using increasingly more concise manners. However, can we do better using logical operators?

λ 17 Xors contd.

```
xor6, xor7, xor8, xor9 :: Bool → Bool → Bool
-- Litrally just using the definition
xor6 b1 b2 = (b1 && (not b2)) || ((not b1) && b2)

-- Recall that the comparision operators return bools?
xor7 b1 b2 = b1 ≠ b2

-- And using the fact that operators are functions..
xor8 b1 b2 = (≠) b1 b2

-- .. we can have a 4 character definition.
xor9 = (≠)
```

We could also use `if .. then .. else` syntax. To jog your memory, the `if` keyword is followed by some condition, aka a function that returns `True` or `False`, this is followed by the `then` keyword and a function to excute if the condition is satisfied and the `else` keyword and a function to execute as a if the condition is not satisfied. For example

λ 17 Xors, contd.

```
xor10, xor11 :: Bool → Bool → Bool

xor10 b1 b2 = if b1 == b2 then False else True

xor11 b1 b2 = if b1 ≠ b2 then True else False
```

Or use the guard syntax. Similar to piecewise functions in math, we can define the function piecewise with the input changing the definition of the function, we can define guarded definition where the inputs control which definition we access. If the pattern(a condition) to a guard is met, that definition is accessed in order of declaration.

We do this as follows

λ 17 Xors, cotd

```
xor12, xor13, xor14, xor15 :: Bool → Bool → Bool

xor12 b1 b2
  | b1 == True = not b2 -- If b1 is True, the code acesses this definition
regardless of b2's value. The function enters the definition which matches
first.
  | b2 == False = b1

-- Can you spot a problem in xor12? xor12 False True is not defined and
would raise the exception Non-exhaustive patterns in function xor12.
-- This means that the pattern of inputs provided can't match with any of
the definitions. We can fix it by either being careful and matching all
the cases..

xor13 False b2 = b2 -- Notice, we can have part of the definition
unguarded before entering the guards.
xor13 True b2
  | b2 == False = True
  | b2 == True = False

xor14 b1 b2
  | b1 == b2 = False
  | b1 ≠ b2 = True

-- .. or by using the otherwise keyword, we can define a catch-all case.
If none of the patterns are matched, the function enters the otherwise
definiton.

xor15 b1 b2
  | b1 == True = not b2
  | otherwise = b2
```

Finally, we can define use the `case .. of ..` syntax. While this syntax is rarer, and too verbose, for simple functions, we will see a lot of it later in [monads chapter]. In this syntax, the general form is

```
case <expression> of
  <pattern1> → <result1>
  <pattern2> → <result2>
  ...
```

The case expression evaluates the `<expression>`, and matches it against each pattern in order. The first matching pattern's corresponding result is returned. You can nest case expressions to match on multiple values, although it can become extreamly unreadable, rather quickly.

λ 17 Xors, contd

```haskell
xor16, xor17 :: Bool → Bool → Bool

-- We use a single case on the first input.
xor16 :: Bool → Bool → Bool
xor16 b1 b2 = case b1 of
  False → b2
  True → not b2

-- Or we can return to defining for every single case, just using more
words.
xor17 b1 b2 = case b1 of
  False → case b2 of
    False → False
    True → True
  True → case b2 of
    False → True
    True → False
```

Now that we are done with this tiresome activity, and learned a lot of Haskell syntax, let's go for a ride.

X Exercise

It is a well know fact that one can define all logical operators using only `nand`. Well, let's do so. Redefine `and, or, not, xor` using only `nand`.

Numerical Functions

A lot of numeric operators and functions come predefined in Haskell. Some natural ones are

```
>>> 7 + 3
10
>>> 3 + 8
11
>>> 97 + 32
129
>>> 3 - 7
-4
>>> 5 - (-6)
11
>>> 546 - 312
234
>>> 7 * 3
21
>>> 8*4
32
>>> 45 * 97
4365
>>> 45 * (-12)
-540
>>> (-12) * (-11)
132
>>> abs 10
10
>>> abs (-10)
10
```

The internal definition of addition and subtraction is discussed in the appendix while we talk about some multiplication algorithms in the time complexity chapter. For our purposes, we want it to be clear and predictable what one expects to see when any of these operators are used. `Abs` is also implemented in a very simple fashion.

λ Implementation of abs function

```
abs :: Num a ⇒ a → a
abs a = if a ⩾ 0 then a else -a
```

Division, A Trilogy

Now let's move to the more interesting operators and functions.

`recip` is a function which reciptocates a given numebr, but it has rather interesting type signature. It is only defined on types with the `Fractional` typeclass. This refers to a lot of things, but the most common ones are `Rational`, `Float` and `Double`. `recip`, as the name suggests, returns the reciprocal of the number taken as input. The type signature is `recip :: Fractional a ⇒ a → a`

```
>>> recip 5
0.2
>>> k = 5 :: Int
>>> recip k
<interactive>:47:1: error: [GHC-39999] ...
```

It is clear that in the above case, 5 was treated as a `Float` or `Double` and the expected output provided. In the following case, we specified the type to be `Int` and it caused a horrible error. This is because for something to be a fractional type, we literally need to define how to reciprocate it. We will talk about how exactly it is defined in < some later chapter probably 8 >. For now, once we have `recip` defined, division can be easily defined as

```
(/) :: Fractional a ⇒ a → a → a
x / y = x * (recip y)
```

Again, notice the type signature of `(/)` is `Fractional a ⇒ a → a → a`.[3]

However, this is not the only division we have access to. Say we want only the quotient, then we have `div` and `quot` functions. These functions are often coupled with `mod` and `rem` are the respective remainder functions. We can get the quotient and remainder at the same time using `divMod` and `quotRem` functions. A simple example of usage is

[3] It is worth pointing out that one could define `recip` using `(/)` as well given 1 is defined. While this is not standard, if `(/)` is defined for a data type, Haskell does autmoatically infer the reciprocation. So technically, for a datatype to be a memeber of the type class `Fractional` it needs to have either reciprocation or division defined, the other is infered.

```
>>> 100 `div` 7
14
>>> 100 `mod` 7
2
>>> 100 `divMod` 7
(14,2)
>>> 100 `quot` 7
14
>>> 100 `rem` 7
2
>>> 100 `quotRem` 7
(14,2)
```

One must wonder here that why would we have two functions doing the same thing? Well, they don't actually do the same thing.

Exercise

From the given example, what is the difference between `div` and `quot`?

```
>>> 8 `div` 3
2
>>> (-8) `div` 3
-3
>>> (-8) `div` (-3)
2
>>> 8 `div` (-3)
-3
>>> 8 `quot` 3
2
>>> (-8) `quot` 3
-2
>>> (-8) `quot` (-3)
2
>>> 8 `quot` (-3)
-2
```

Exercise

From the given example, what is the difference between `mod` and `rem`?

```
>>> 8 `mod` 3
2
>>> (-8) `mod` 3
1
>>> (-8) `mod` (-3)
-2
>>> 8 `mod` (-3)
-1
>>> 8 `rem` 3
2
>>> (-8) `rem` 3
-2
>>> (-8) `rem` (-3)
-2
>>> 8 `rem` (-3)
2
```

While the functions work similerly when the divisior and dividend are of the same sign, they seem to diverge when the signs don't match. The thing here is we ideally want our division algorithm to satisfy $d * q + r = n, |r| < |d|$ where d is the divisior, n the dividend, q the quotient and `r` the remainder. The issue is for any $-d < r < 0 \Rightarrow 0 < r < d$. This means we need to choose the sign for the remainder.

In Haskell, `mod` takes the sign of the divisor(comes from floored division, same as Python's `%`), while `rem` takes the sign of the dividend (comes from truncated division, behaves the same way as Scheme's `remainder` or C's `%`.).

Basically, `div` returns the floor of the true divison value(recall $\lfloor -3.56 \rfloor = -4$) while `quot` returns the trunicated value of the true division(recall $\text{trunicate}(-3.56) = -3$ as we are just trunicating the decimal point off). The reason we keep both of them in Haskell is to be comfertable for people who come from either of these languages. Also, The `div` function is often the more natural one to use, whereas the `quot` function corresponds to the machine instruction on modern machines, so it's somewhat more efficient(although not much, I had to go upto 10^{100000} to even get millisecond difference in the two).

A simple excercise for us now would be implementing our very own integer division algorithm. We begin with a division algorithm for only positive integers.

λ A division algorithm on positive integers by repreated subtraction

```
divide :: Integer → Integer → (Integer, Integer)
divide n d = go 0 n where
  go q r = if r ⩾ d then go (q+1) (r-d) else (q,r)
```

Now, how do we extend it to negitives by a little bit of case handling.

```haskell
divideComplete :: Integer → Integer → (Integer, Integer)
divideComplete _ 0 = error "DivisionByZero"
divideComplete n d
  | d < 0     = let (q, r) = divideComplete n (-d) in (-q, r)
  | n < 0     = let (q, r) = divideComplete (-n) d in if r == 0 then (-q,
0) else (-q - 1, d - r)
  | otherwise = divideUnsigned n d

divide :: Integer → Integer → (Integer, Integer)
divide n d = go 0 n where
  go q r = if r ⩾ d then go (q+1) (r-d) else (q,r)
```

An excercise left for the reader is to figure out which kind of division is this, floored or trunicated, and implement the one we haven't yourself. Let's now tal

Exponentiation

Haskell defines for us three exponation operators, namely `(^^), (^), (**)`.

Exercise

What can we say about the three exponation operators?

```haskell
<will make this example later>
```

Unlike division, they have almost the same function. The difference here is in the type signature. While, infering the exact type signature was not expected, we can notice:

- `^` is raising genral numbers to positive integral powers. This means it makes no assumptions about if the base can be reciprocated and just produces an error if the power is negative.
- `^^` is raising fractional numbers to general integral powers. That is, it needs to be sure that the reciprocal of the base exists(negative powers) and doesn't throw an error if the power is negative.
- `**` is raising numbers with floating point to powers with floating point. This makes it the most general exponation.

The operators clearly get more and more general as we go down the list but they also get slower. However, they are also reducing in accurecy and may even output `Infinity` in some cases. The `...` means I am trunicating the output for readablity, ghci did give the compelete answer.

```
>>> 2^1000
10715086071862673209484250490600018105614048117055336074 ...
>>> 2 ^^ 1000
1.0715086071862673e301
>>> 2^10000
19950631168807583848837421626835850838 2 ...
>>> 2^^10000
Infinity
>>> 2 ** 10000
Infinity
```

The exact reasons for the inaccuracy comes from float conversions and approximation methods. We will talk very little about this specialist topic somewhat later.

However, something within our scope is implementing `(^)` ourselves.

λ A naive integer exponation algorithm

```haskell
exponation :: (Num a, Integral b) ⇒ a → b → a
exponation a 0 = 1
exponation a b = if b < 0
  then error "no negitve exponation"
  else a * (exponation a (b-1))
```

This algorithm, while the most naive way to do so, computes 2^{100000} in mearly 0.56 seconds.

However, we could do a bit better here. Notice, to evaluate a^b, we are making b multiplications. A fact we mentioned before is that multiplication of big numbers is faster when it is balenced, that is the numbers being multiplied have similer number of digits.

So to do better, we could simply compute $a^{\frac{b}{2}}$ and then square it, given b is even, or compute $a^{\frac{b-1}{2}}$ and then square it and multiply by a otherwise. This can be done recusrsivly till we have the solution.

λ A better exponentiation algorithm using divide and conquer

```haskell
exponation :: (Num a, Integral b) ⇒ a → b → a
exponation a 0 = 1
exponation a b
  | b < 0     = error "no negitve exponation"
  | even b    = let half = exponation a (b `div` 2)
                in half * half
  | otherwise = let half = exponation a (b `div` 2)
                in a * half * half
```

The idea is simple: instead of doing b multiplications, we do far fewer by solving a smaller problem and reusing the result. While one might not notice it for smaller b's, once we get into the hundreds or thousands, this method is dramatically faster.

This algorithm brings the time to compute 2^{100000} down to 0.07 seconds.

The idea is that we are now making atmost 3 multiplications at each step and there are atmost $\log(b)$ steps. This brings us down from b multiplications to $3\log(b)$ multiplications. Furthermore, most of these multiplications are somewhat balenced and hence optimized.

This kind of a stratergy is called divide and conquer. You take a big problem, slice it in half, solve the smaller version, and then stitch the results together. It's a method/technique that appears a lot in Computer Science(in sorting to data search to even solving diffrential equations and training AI models) and we will see it again shortly.

Finally, there's one more minor optimization that's worth pointing out. It's a small thing, and doesn't even help that much in this case, but if the multiplication were particularly costly, say as in matrices; our exponation method could be made slightlty better. Let's say we are dealing with say 2^{255}. Our current algorithm would evaluate it as:

$$
\begin{aligned}
2^{31} &= \left(2^{15}\right)^2 * 2 \\
&= \left(\left(2^7\right)^2 * 2\right)^2 * 2 \\
&= \left(\left(\left(2^3\right)^2 * 2\right)^2 * 2\right)^2 * 2 \\
&= \left(\left(\left(\left(2^1\right)^2 * 2\right)^2 * 2\right)^2 * 2\right)^2 * 2
\end{aligned}
$$

This is a problem as the small $* 2$ in every bracket are unbalenced. The exact way we deal with all this is by something called 2^k arry method. Although, more often then not, most built in implementations use the divide and conquer exponentiation we studied.

`gcd` and `lcm`

A very common function for number theoretic use cases is `gcd` and `lcm`. They are pre-defined as

```
>>> :t gcd
gcd :: Integral a ⇒ a → a → a
>>> :t lcm
lcm :: Integral a ⇒ a → a → a
>>> gcd 12 30
6
>>> lcm 12 30
60
```

We will now try to define these functions ourselves.

A naive way to do so would be:

λ Naive GCD and LCM

```
-- Uses a brute-force approach starting from the smaller number and
counting down
gcdNaive :: Integer → Integer → Integer
gcdNaive a 0 = a
gcdNaive a b =
    if b > a
        then gcdNaive b a  -- Ensure first argument is greater
        else go a b b
  where
    -- Start checking from the smaller of the two numbers
    go x y current =
        if (x `mod` current == 0) && (y `mod` current == 0)
            then current
            else go x y (current - 1)

-- Uses a brute-force approach starting from the larger number and
counting up
lcmNaive :: Integer → Integer → Integer
lcmNaive a b =
    if b > a
        then lcmNaive b a  -- Ensure first argument is greater
        else go a b a
  where
    -- Start checking from the larger of the two numbers
    go x y current =
        if current `mod` y == 0
            then current
            else go x y (current + x)
```

These both are quite slow for most practical uses. A lot of cryptography runs on computer's ability to find gcd and lcm fast enough. If this was the fastest, we would be cooked. So what do we do? Call some math.

A simple optimization could be using $p * q = \gcd(p, q) * \text{lcm}(p, q)$. This makes the speed of both the operations same, as once we have one, we almost already have the other.

Let's say we want to find $g := \gcd(p, q)$ and $p > q$. That would imply $p = dq + r$ for some $r < q$. This means $g \mid p, q \Rightarrow g \mid q, r$ and by the maximality of g, $\gcd(p, q) = \gcd(q, r)$. This helps us out a lot as we could eventually reduce our problem to a case where the larger term is a multiple of the smaller one and we could return the smaller term then and there. This can be implemented as:

λ Fast GCD and LCM

```haskell
gcdFast :: Integer → Integer → Integer
gcdFast p 0 = p -- Using the fact that the moment we get q | p, we will
reduce to this case and output the answer.
gcdFast p q = gcdFast q (p `mod` q)


lcmFast :: Integer → Integer → Integer
lcmFast p q = (p * q) `div` (gcdFast p q)
```

We can see that this is much faster. The exact number of steps or time taken is a slightlty involved and not very related to what we cover. Intrested readers may find it and related citrations here.

This algorithm predates computers by approximatly 2300 years. If was first decribed by Euclid and hence is called the Euclidean Algorithm. While, faster algorithms do exist, the ease of implementation and the fact that the optimizations are not very dramatic in speeding it up make Euclid the most commonly used algorithm.

While we will see these class of algorithms, including checking if a number is prime or finding the prime factorization, these require some more weapons of attack we are yet to devlop.

Recursive Functions

A lot of mathematical functions are defined recusrsivly. We have already seen a lot of them in < chapter 1>. Factorial, binomials and fibbonacci are common examples. We will implement them here for the the sake of completness, although I don't think converting them from paper to code is hard, we will still do it.

λ Factorial, Binomial and Fibbonacci

```haskell
factorial :: Integer → Integer
factorial 0 = 1
factorial n = n *  factorial (n-1)

nCr :: Integer → Integer → Integer
nCr _ 0 = 1
nCr n r
  |r > n      = 0
  |n == r     = 1
  |otherwise  = (nCr (n-1) (r-1)) + (ncr (n-1) r)

fibbonacci :: Integer → Integer
fibbonacci n = fst (go n) where
  go 0 = (1,0)
  go 1 = (1, 1)
  go n = (a + b , a) where (a,b) = go (n-1)
```

You might remember that we don't directly translate the defination of fibbonacci as doing so would be extreamly inafficent, as we would be recomputing values left and right. A much simpler way is to carry the data we need. And that is what we do here.

Mathematical Functions

We will now talk about mathematical functions like `log`, `sqrt`, `sin`, `asin` etc. We will also take this oppurtunity to talk about real exponation. To begin, Haskell has a lot of pre-defined functions.

```
>>> sqrt 81
9.0

>>> log (2.71818)
0.9999625387017254
>>> log 4
1.3862943611198906
>>> log 100
4.605170185988092
>>> logBase 10 100
2.0
>>> exp 1
2.718281828459045
>>> exp 10
22026.465794806718

>>> pi
3.141592653589793
>>> sin pi
1.2246467991473532e-16
>>> cos pi
-1.0
>>> tan pi
-1.2246467991473532e-16
>>> asin 1
1.5707963267948966
>>> asin 1/2
0.7853981633974483
>>> acos 1
0.0
>>> atan 1
0.7853981633974483
```

`pi` is a predefined variable inside haskell. It carries the value of π upto some decimal places based on what type it is forced in.

```
>>> a = pi :: Float
>>> a
3.1415927
>>> b = pi :: Double
>>> b
3.141592653589793
```

All the fucntions above have the type signature `Fractional a ⇒ a → a` or for our purposes `Float → Float`. Also, notice the functions are not giving exact answers in some cases and instead are giving approximations. These functions are quite unnatural for a computer, so we surely know that the computer isn't processing them. So what is happening under the hood?

Imagine you're playing a number guessing game with a friend.

They are thinking of a number between 1 and 100, and every time you guess, they'll say whether your guess is too high, too low, or correct.

You don't start at 1. You start at 50. Why? Because 50 cuts the range exactly in half. Depending on whether the answer is higher or lower, you can now ignore half the numbers.

Next guess? Halfway through the remaining half. Then half of that. And so on.

That's binary search: each step cuts the list in half, so you zoom in on the answer quickly.

Here's how it works:

- Start in the middle of a some ordered list.
- If the middle item is your target, you're done.
- If it's too big, repeat the search on the left half.
- If it's too small, repeat on the right half.

Keep halving until you find it - or realize it's not there.

While using a raw binery search for roots would be impossible as the exact answer is seldom rational and hence, the algorithm would never terminate. So instead of searching for the exact root, we look for an approximation by keeping some tolerence. Here is what it looks like:

λ Square root by binary search

```
bsSqrt :: Float → Float → Float
bsSqrt tolerance n
  | n > 1     = binarySearch 1 n
  | otherwise = binarySearch 0 1
  where
    binarySearch low high
      | abs (guess * guess - n) ⩽ tolerance        = guess
      | guess * guess > n                          = binarySearch low
guess
      | otherwise                                  = binarySearch guess
high
      where
        guess = (low + high) / 2
```

We leave it as an excercise to extend this to a cube root.

The internal implementation sets the tolerance to some constant, defining, for example as `sqrt = bsSqrt 0.00001`

Furthermore, there is a faster method to compute square roots and cube roots(in general roots of polynomials), which uses a bit of analysis. You will find it defined and walked-through in the back excercise.

However, this method won't work for `log` as we would need to do real exponation, which, as we will soon see, is defined using `log`. So what do we do? Taylor series and reduction.

We know that $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$. For small x, $\ln(1+x) \approx x$. So if we can create a scheme to make x small enough, we could get the logithrm by simply multiplying. Well, $\ln(x^2) = 2\ln(|x|)$. So, we could simply keep taking square roots of a number till it is within some error range of 1 and then simply use the fact $\ln(1+x) \approx x$ for small x.

λ Log defined using Taylor Approximation

```
logTay :: Float → Float → Float
logTay tol n
  | n ⩽ 0                    = error "Negative log not defined"
  | abs(n - 1) ⩽ tol         = n - 1  -- using log(1 + x) ≈ x
  | otherwise                = 2 * logTay tol (sqrt n)
```

This is a very efficient algorithm for approximating `log`. Doing better requires the use of either pre-computed lookup tables(which would make the programme heavier) or use more sophesticated mathematical methods which while more accurate would slow the programme down. There is an excercise in the back, where you will implement a state of the art algorithm to compute `log` accurately upto 400-1000 decimal places.

Finally, now that we have `log = logTay 0.0001`, we can easily define some other functions.

```
logBase a b = log(b) / log(a)
exp n = if n == 1 then 2.71828 else (exp 1) ** n
(**) a b = exp (b * log(a))
```

We will use this same Taylor approximation scheme for `sin` and `cos`. The idea here is: $\sin(x) \approx x$ for small x and $\cos(x) = 1$ for small x. Furthermore, $\sin(x + 2\pi) = \sin(x)$, $\cos(x + 2\pi) = \cos(x)$ and $\sin(2x) = 2\sin(x)\cos(x)$ as well as $\cos(2x) = \cos^2(x) - \sin^2(x)$.

This can be encoded as

λ Sin and Cos using Taylor Approximation

```
sinTay :: Float → Float → Float
sinTay tol x
  | abs(x) ⩽ tol         = x  -- Base case: sin(x) ≈ x when x is small
  | abs(x) ⩾ 2 * pi      = if x > 0
                           then sinTay tol (x - 2 * pi)
                           else sinTay tol (x + 2 * pi)  -- Reduce x to
[-2π, 2π]
  | otherwise            = 2 * (sinTay tol (x/2)) * (cosTay tol (x/2))  --
sin(x) = 2 sin(x/2) cos(x/2)

cosTay :: Float → Float → Float
cosTay tol x
  | abs(x) ⩽ tol         = 1.0  -- Base case: cos(x) ≈ 1 when x is small
  | abs(x) ⩾ 2 * pi      = if x > 0
                           then cosTay tol (x - 2 * pi)
                           else cosTay tol (x + 2 * pi)  -- Reduce x to
[-2π, 2π]
  | otherwise            = (cosTay tol (x/2))**2 - (sinTay tol (x/2))**2
-- cos(x) = cos²(x/2) - sin²(x/2)
```

As one might notice, this approximation is somewhat poorer in accuracy than `log`. This is due to the fact that the taylor approximation is much less truer on `sin` and `cos` in the neighbourhood of `0` than for `log`.

We will see a better approximation once we start using lists, using the power of the full Taylor expansion.

Finally, similer to our above things, we could simply set the tolerance and get a function that takes an input and gives an output, name it `sin` and `cos` and define `tan x = (sin x) / (cos x)`.

It is left as excercise to use taylor approximation to define inverse sin(`asin`), inverse cos(`acos`) and inverse tan(`atan`).

Collatz

Collatz conjucture states that for any $n \in \mathbb{N}$ exists a k such that $c^{k(n)} = 1$ where c is the Collatz function which is $\frac{n}{2}$ for even n and $3n + 1$ for odd n.

Write a functuon `col :: Integer → Integer` which, given a n, finds the smalltest k such that $c^{k(n)} = 1$, called the Collatz chain length of n.

Newton–Raphson method

Newton–Raphson method

Newton–Raphson method is a method to find the roots of a function via subsequent approximations.

Given $f(x)$, we let x_0 be an inital guess. Then we get subsequent guesses using

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

As $n \to \infty$, $f(x_n) \to 0$.

The intution for why this works is: imagine standing on a curve and wanting to know where it hits the x-axis. You draw the tangent line at your current location and walk down it to where it intersects the x-axis. That's your next guess. Repeat. If the curve behaves nicely, you converge quickly to the root.

Limitations of Newton–Raphson method are

- Requires derivative: The method needs the function to be differentiable and requires evaluation of the derivative at each step.
- Initial guess matters: A poor starting point can lead to divergence or convergence to the wrong root.
- Fails near inflection points or flat slopes: If $f'(x)$ is zero or near zero, the method can behave erratically.
- Not guaranteed to converge: Particularly for functions with multiple roots or discontinuities.

Considering, $f(x) = x^2 - a$ and $f(x) = x^3 - a$ are well behaved for all a, implement `sqrtNR :: Float → Float → Float` and `cbrtNR :: Float → Float → Float` which finds the square root and cube root of a number upto a tolerance using the Newton–Raphson method.

Hint: The number we are trying to get the root of is a sufficiently good guess for numbers absolutly greater than 1. Otherwise, 1 or -1 is a good guess. We leave it to your mathematical intution to figure out when to use what.

Digital Root

The digital root of a number is the digit obtained by summing digits until you get a single digit. For example `digitalRoot 9875 = digitalRoot (9+8+7+5) = digitalRoot 29 = digitalRoot (2+9) = digitalRoot 11 = digitalRoot (1+1) = 2`
Implement the function `digitalRoot :: Int → Int`.

AGM Log

A rather uncommon mathematical function is AGM or arthmatic-geometric mean. For given two numbers,

$$\mathrm{AGM}(x,y) = \begin{cases} x & \text{if } x = y \\ \mathrm{AGM}\left(\frac{x+y}{2}, \sqrt{xy}\right) & \text{otherwise} \end{cases}$$

Write a function `agm :: (Float, Float) → Float → Float` which takes two floats and returns the AGM within some tolerance(as getting to the exact one recusrsivly takes, about infinite steps).

Using AGM, we can define

$$\ln(x) \approx \frac{\pi}{2\,\mathrm{AGM}\left(1, \frac{2^{2-m}}{x}\right)} - m\ln(2)$$

which is precise upto p bits where $x2^m > 2^{\frac{p}{2}}$.

Using the above defined `agm` function, define `logAGM :: Int → Float → Float → Float` which takes the number of bits of precision, the tolerance for `agm` and a number greater than 1 and gives the natural logithrm of that number.

Hint: To simplify the question, we added the fact that the input will be greater than 1. This means a simplification is taking `m = p/2` directly. While geting a better `m` is not hard, this is just simpler.

X Multiplexer

A multiplexer is a hardware element which chooses the input stream from a variety of streams.

It is made up of $2^n + n$ components where the 2^n are the input streams and the n are the selectors.

(i) Implement a 2 stream multiplex `mux2 :: Bool → Bool → Bool → Bool` where the first two booleans are the inputs of the streams and the third boolean is the selector. When the selector is `True`, take input from stream 1, otherwise from stream 2.

(ii) Implement a 2 stream multiplex using only boolean operations.

(iii) Implement a 4 stream multiplex. The type should be `mux4 :: Bool → Bool → Bool → Bool → Bool → Bool → Bool`. (There are 6 arguments to the function, 4 input streams and 2 selectors). We encourage you to do this in atleast 2 ways (a) Using boolean operations (b) Using only `mux2`.

Could you describe the general scheme to define `mux2^n` (a) using only boolean operations (b) using only `mux2^(n-1)` (c) using only `mux2`?

X Moduler Exponation

Implement modular exponentiation $(a^b \bmod m)$ efficiently using the fast exponentiation method. The type signature should be `modExp :: Int → Int → Int → Int`

Types as Sets

Ryan Hota

Sets

set

A **set** is a ***well-defined collection of "things"***.
These "things" can be values, objects, or other sets.
For any given set, the "things" it contains are called its **elements**.

Some basic kinds of sets are -

- **empty set**

  The **empty set** is the ***set that contains no elements*** or equivalently, $\{\}$.

- **singleton set**

  A **singelten set** is a ***set that contains exactly one element***, such as $\{34\}$, $\{\triangle\}$, the set of natural numbers strictly betweeen 1 and 3, etc.

We might have encountered some mathematical sets before, such as the set of real numbers $\mathbb{R}$ or the set of natural numbers $\mathbb{N}$, or even a set following the rules of vectors (a vector space).

We might have encountered sets as data structures acting as an unordered collection of objects or values, such as Python sets - `set([])`,`{1,2,3}`, etc.

Note that sets can be finite ($\{12, 1, \circ, \vec{x}\}$), as well as infinite ($\mathbb{N}$) .

A fundamental keyword on sets is "$\in$", or "belongs".

belongs

Given a value x and a set S,
$x \in S$ is a ***claim*** that x ***is an element of*** S,

Other common operations include -

union

$A \cup B$ is the ***set containing all those*** x ***such that either*** $x \in A$ ***or*** $x \in B$.

intersection

$A \cap B$ is the ***set containing all those*** x ***such that*** $x \in A$ ***and*** $x \in B$.

cartesian product

$A \times B$ is the ***set containing all ordered pairs*** (a, b) ***such that*** $a \in A$ ***and*** $b \in B$.

So,

$$X == \{x_1, x_2, x_3\} \text{ and } Y == \{y_1, y_2\}$$
$$\Rightarrow$$
$$X \times Y == \{(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2), (x_3, y_1), (x_3, y_2)\}$$

set exponent

B^A is the ***set of all functions with domain*** A ***and co-domain*** B,
or equivalently , the ***set of all functions*** f ***such that*** $f : A \to B$,
or equivalently , the ***set of all functions from*** A ***to*** B.

X size of exponent set

If A has $|A|$ elements, and B has $|B|$ elements, then how many elements does B^A have?

Types

We have enconutered a few types in the previous chapter, such as `Bool`, `Integer` and `Char`. For our limited purposes, we can think about each such **type** as the **set of all values of that type**.

For example,

- `Bool` can be thought of as the **set of all bolean values**, which is {`False`, `True`}.
- `Integer` can be thought of as the **set of all integers**, which is {`0`, `1`, `-1`, `2`, `-2`, . . . }.
- `Char` can be thought of as the **set of all characters**, which is {`'\NUL'`, `'\SOH'`, `'\STX'`, . . . , `'a'`, `'b'`, `'c'`, . . . , `'A'`, `'B'`, `'C'`, . . . }

If this analogy were to extend further, we might expect to see analogues of the basic kinds of sets and the common set operations for types, which we can see in the following -

`::` is analogous to $\in$ or ÷ belongs

Whenever we want to claim a value `x` is of type `T`, we can use the `::` keyword, in a similar fashion to $\in$, i.e., we can say `x :: T` in place of $x \in T$.

In programming terms, this is known as declaring the variable `x`.

For example,

- λ declaration of x

```
x :: Integer
x = 42
```

  This reads - "Let $x \in \mathbb{Z}$. Take the value of x to be 42."

- λ declaration of y

```
y :: Bool
y = xor True False
```

  This reads - "Let $y \in$ {False, True}. Take the value of y to be the $\oplus$ of True and False."

X declaring a variable

Declare a variable of type `Char`.

`A → B` is analogous to B^A or ÷ set exponent

As B^A contains all functions from A to B,
so is each function `f` defined to take an input of type `A` and output of type `B` satisfy `f :: A→B`.

For example -

- λ function

```
succ :: Integer → Integer
succ x = x + 1
```

- λ another function

```
even :: Integer → Bool
even n = if n `mod` 2 == 0 then True else False
```

X basic function definition

Define a non-constant function of type `Bool → Integer`.

X difference between declaration and function definition

What are the differences between declaring a variable and defining a function?

`( A , B )` is analogous to $A \times B$ or ÷ cartesian product

As $A \times B$ contains all pairs (a, b) such that $a \in A$ and $b \in B$,
so is every pair `(a,b)` of type `(A,B)` if `x` is of type `A` and `b` is of type `B`.

For example, if I ask GHCi to tell me the type of `(True, 'c')` (which I can do using the command `:t`), then it would tell me that the value's type is `(Bool, Char)` -

λ type of a pair

```
>>> :t (True, 'c')
(True, 'c') :: (Bool, Char)
```

This reads - "GHCi, what is the type of `(True, 'c')`?
Answer : the type of `(True, 'c')` is `(Bool, Char)`."

If we have a type `X` with elements `X1`, `X2`, and `X3`, and another type `Y` with elements `Y1` and `Y2`, we can use the author-defined function `listOfAllElements` to obtain a list of all elements of certain types -

λ elements of a product type

```
>>> listOfAllElements :: [X]
[X1,X2,X3]

>>> listOfAllElements :: [Y]
[Y1,Y2]

>>> listOfAllElements :: [(X,Y)]
[(X1,Y1),(X1,Y2),(X2,Y1),(X2,Y2),(X3,Y1),(X3,Y2)]

>>> listOfAllElements :: [(Char,Bool)]
[('\NUL',False),('\NUL',True),('\SOH',False),('\SOH',True), . . . ]
```

There are two fundamental inbuilt operations from a product type -

A function to get the first component of a pair -

λ first component of a pair

```
fst (a,b) = a
```

and a similar function to get the second component -

λ second component of a pair

```
snd (a,b) = b
```

We can define our own functions from a product type using these -

λ function from a product type

```
xorOnPair :: ( Bool , Bool ) → Bool
xorOnPair pair = ( fst pair ) ≠ ( snd pair )
```

or even by pattern matching the pair -

λ another function from a product type

```
xorOnPair' :: ( Bool , Bool ) → Bool
xorOnPair' ( a , b ) = a ≠ b
```

Also, we can define our functions to a product type -
For example, consider the useful inbuilt function `divMod`, which **divides a number by another**, and **returns** both the **quotient and the remainder as a pair**. Its definition is equivalent to the following -

λ function to a product type

```
divMod :: Integer → Integer → ( Integer , Integer )
divMod n m = ( n `div` m , n `mod` m )
```

x size of a product type

If a type `T` has n elements, and type `T'` has m elements, then how many elements does `(T,T')` have?

`()` is analogous to ÷ singleton set

`()`, pronounced Unit, is a type that contains exactly one element.
That unique element is `()`.

So, it means that `()::()`, which might appear a bit confusing.

The `()` on the left of `::` is just a simple value, like `1` or `'a'`.
The `()` on the right of `::` is a type, like `Integer` or `Char`.

This value `()` is the only value whose type is `()`.

On the other hand, other types might have multiple values of that type. (such as `Integer`, where both `1` and `2` have type `Integer`.)

We can even check this using `listOfAllElements` -

```
>>> listOfAllElements :: [()]
[()]
```

This reads - "The list of all elements of the type `()` is a list containing exactly one value, which is the value `()`."

x function to unit

Define a function of type `Bool → ()`.

x function from unit

Define a function of type `() → Bool`.

No ÷ intersection of Types

We now need to discuss an important distinction between sets and types. While two different sets can have elements in common, like how both $\mathbb{R}$ and $\mathbb{N}$ have the element 10 in common, on the other hand, two different types `T1` and `T2` cannot have any common elements.

For example, the types `Int` and `Integer` have no elements in common. We might think that they have the element `10` in common, however, the internal structures of `10 :: Int` and `10 :: Integer` are very different, and thus the two `10`s are quite different.

Thus, the intersection of two different types will always be empty and doesn't make much sense anyway.

Therfore, no intersection operation is defined for types.

No ÷ union of Types

Suppose the type `T1 ∪ T2` were an actual type. It would have elements in common with the type `T1`. As discussed just previously, this is undesirable and thus disallowed.

But there is a promising alternative, for which we need to define the set-theoretic notion of **disjoint union**.

x **subtype**

Do you think that there can be an anlogue of the *subset* relation $\subseteq$ for types?

Disjoint Union of Sets

÷ **disjoint union**

$A \sqcup B$ is defined to be $(\{0\} \times A) \cup (\{1\} \times B)$, or equivalently, *the set of all pairs either of the form $(0, a)$ such that $a \in A$, or of the form $(1, b)$ such that $b \in B$*.

So,

$$X == \{x_1, x_2, x_3\} \text{ and } Y == \{y_1, y_2\}$$
$$\Rightarrow$$
$$X \sqcup Y == \{(0, x_1), (0, x_2), (0, x_3), (1, y_1), (1, y_2)\}$$

The main advantage that this construct offers us over the usual ÷ union is that given an element x from a disjoint union $A \sqcup B$, it is very easy to see whether x comes from A, or whether it comes from B.

For example, consider the statement - $(0, 10) \in \mathbb{R} \sqcup \mathbb{N}$.
It is obvious that this 10 comes from $\mathbb{R}$ and does not come from $\mathbb{N}$.
$(1, 10) \in \mathbb{R} \sqcup \mathbb{N}$ would indicate exactly the opposite, i.e, the 10 here comes from $\mathbb{N}$, not $\mathbb{R}$.

`Either A B` is analogous to $A \sqcup B$ or ÷ disjoint union

The term "either" is motivated by its appearance in the definition of ÷ disjoint union.

Recall that in a ÷ disjoint union , each element has to be
- of the form $(0, a)$, where $a \in A$, and A is the set to the left of the $\sqcup$ symbol,

- or they can be of the form $(1, b)$, where $b \in B$, and B is the set to the right of the $\sqcup$ symbol.

Similarly, in `Either A B`, each element has to be

- of the form `Left a`, where `a :: A`
- or of the form `Right b`, where `b :: B`

If we have a type `X` with elements `X1`, `X2`, and `X3`, and another type `Y` with elements `Y1` and `Y2`, we can use the author-defined function `listOfAllElements` to obtain a list of all elements of certain types -

λ elements of an either type

```
>>> listOfAllElements :: [X]
[X1,X2,X3]

>>> listOfAllElements :: [Y]
[Y1,Y2]

>>> listOfAllElements :: [Either X Y]
[Left X1,Left X2,Left X3,Right Y1,Right Y2]

>>> listOfAllElements :: [Either Bool Char]
[Left False,Left True,Right '\NUL',Right '\SOH',Right '\STX', . . . ]
```

We can define functions to an `Either` type.
Consider the following problem : We have to make a function that provides feedback on a quiz. We are given the marks obtained by a student in the quiz marked out of 10 total marks. If the marks obtained are less than 3, return `'F'`, otherwise return the marks as a percentage -

λ function to an either type

```
feedback :: Integer → Either Char Integer
--                    Left ~ Char,Integer ~ Right
feedback n
  | n < 3     = Left  'F'
  | otherwise = Right ( 10 * n ) -- multiply by 10 to get percentage
```

This reads - “

Let `feedback` be a function that takes an `Integer` as input and returns `Either` a `Char` or an `Integer`.

As `Char` and `Integer` occurs on the left and right of each other in the expression `Either Char Integer`, thus `Char` and `Integer` will henceforth be referred to as `Left` and `Right` respectively.

Let the input to the function `feedback` be `n`.

If `n<3`, then we return `'F'`. To denote that `'F'` is a `Char`, we will tag `'F'` as `Left`. (remember that `Left` refers to `Char`!)

`otherwise`, we will multiply `n` by `10` to get the percentage out of 100 (as the actual quiz is marked out of 10). To denote that the output `10*n` is an `Integer`, we will tag it with the word `Right`. (remember that `Right` refers to `Integer`!)

“

We can also define a function from an `Either` type.
Consider the folowing problem : We are given a value that is either a boolean or a character. We then have to represent this value as a number.

```
top
import Data.Char(ord)
```

```
λ function from an either type
representAsNumber :: Either Bool Char → Int
--                   Left ~ Bool,Char ~ Right
representAsNumber ( Left  bool ) = if bool then 1 else 0
representAsNumber ( Right char ) = ord char
```

This reads - “

Let `representAsNumber` be a function that takes either a `Bool` or a `Char` as input and returns an `Int`.

As `Bool` and `Char` occurs on the left and right of each other in the expression `Either Bool Char`, thus `Bool` and `Char` will henceforth be referred to as `Left` and `Right` respectively.

If the input to `representAsNumber` is of the form `Left bool`, we know that `bool` must have type `Bool` (as `Left` refers to `Bool`). So if the `bool` is `True`, we will represent it as `1`, else if it is `False`, we will represent it as `0`.

If the input to `representAsNumber` is of the form `Right char`, we know that `char` must have type `Bool` (as `Right` refers to `Char`). So we will represent `char` as `ord char`.

“

We might make things clearer if we use a deeper level of pattern matching, like in the following function (which is equivalent to the last one).

```
λ another function from an either type
representAsNumber' :: Either Bool Char → Int
representAsNumber' ( Left  False ) = 0
representAsNumber' ( Left  True  ) = 1
representAsNumber' ( Right char  ) = ord char
```

X size of an either type

If a type `T` has n elements, and type `T'` has m elements, then how many elements does `Either T T'` have?

The `Maybe` Type

Consider the following problem : We are asked make a function `reciprocal` that reciprocates a rational number, i.e., $\left(x \mapsto \frac{1}{x}\right) : \mathbb{Q} \to \mathbb{Q}$.

Sounds simple enough! Let’s see -

λ naive reciprocal

```
reciprocal :: Rational → Rational
reciprocal x = 1/x
```

But there is a small issue! What about $\frac{1}{0}$?
What should be the output of `reciprocal 0`?

Unfortunately, it results in an error -

```
>>> reciprocal 0
*** Exception: Ratio has zero denominator
```

To fix this, we can do something like this - Let's add one *extra element* to the output type `Rational`, and then `reciprocal 0` can have this *extra element* as its output!

So the new output type would look something like this - ({*extra element*} ⊔ `Rational`)

Notice that this {*extra element*} is a ≒ **singleton set**.
Which means that if we take this *extra element* to be the value `()`,
and take {*extra element*} to be the type `()`,
then we can obtain ({*extra element*} ⊔ `Rational`) as the type `Either () Rational`.

Then we can finally rewrite λ **naive reciprocal** to handle the case of `reciprocal 0` -

λ reciprocal using either

```
reciprocal :: Rational → Either () Rational
reciprocal 0 = Left  ()
reciprocal x = Right (1/x)
```

There is already an inbuilt way to express this notion of `Either () Rational` in Haskell, which is the type `Maybe Rational`.

`Maybe Rational` just names it elements a bit differently compared to `Either () Rational` -

- where
  `Either () Rational` has `Left ()`,
  `Maybe Rational` instead has the value `Nothing`.
- where
  `Either () Rational` has `Right r` (where `r` is any `Rational`),
  `Maybe Rational` instead has the value `Just r`.

Which means that we can rewrite λ **reciprocal using either** using `Maybe` instead -

λ function from a maybe type

```
reciprocal :: Rational → Maybe Rational
reciprocal 0 = Nothing
reciprocal x = Just (1/x)
```

But we can also do this for any arbitrary type `T` in place of `Rational`. In that case -

There is already an inbuilt way to express the notion of `Either () T` in Haskell, which is the type `Maybe T`.

`Maybe T` just names it elements a bit differently compared to `Either () T` -

- where
  `Either () T` has `Left ()`,
  `Maybe T` instead has the value `Nothing`.
- where
  `Either () T` has `Right t` (where `t` is any value of type `T`),
  `Maybe T` instead has the value `Just t`.

If we have a type `X` with elements `X1`, `X2`, and `X3`, and another type `Y` with elements `Y1` and `Y2`, we can use the author-defined function `listOfAllElements` to obtain a list of all elements of certain types -

λ elements of a maybe type

```
>>> listOfAllElements :: [X]
[X1,X2,X3]

>>> listOfAllElements :: [Maybe X]
[Nothing,Just X1,Just X2,Just X3]

>>> listOfAllElements :: [Y]
[Y1,Y2]

>>> listOfAllElements :: [Maybe Y]
[Nothing,Just Y1,Just Y2]

>>> listOfAllElements :: [Maybe Bool]
[Nothing,Just False,Just True]

>>> listOfAllElements :: [Maybe Char]
[Nothing,Just '\NUL',Just '\SOH',Just '\STX',Just '\ETX', . . . ]
```

X size of a maybe type

If a type `T` has n elements, then how many elements does `Maybe T` have?

We can define functions to a `Maybe` type. For example consider the problem of making an inverse function of `reciprocal`, i.e., a function `inversOfReciprocal` s.t.

$$\forall \ \texttt{x :: Rational} \ , \ \texttt{inverseOfReciprocal (reciprocal x) == x}$$

as follows -

λ function to a maybe type

```
inverseOfReciprocal Nothing  = 0
inversOfReciprocal  (Just x) = (1/x)
```

`Void` is analogous to {} or ÷ empty set

The type `Void` has no elements at all.

This also means that no actual value has type `Void`.

Even though it is out-of-syllabus, an interesting exercise is to

X Exercise

try to define a function of type `( Bool → Void ) → Void`.

Introduction to Lists

Ryan Hota

lists (feel free to change it)

Polymorphism and Higher Order Functions

Shubh Sharma

Polymorphism

Functions are our way, to interact with the elements of a type, and one can define functions in one of the two following ways:

1. Define an output for every single element.
2. Consider the general shape and behaviour of elements, and how they interact with other simple functions, then build more complex function with that information.

Up until the chapter about lists, we saw how to define functions from a given type, to another given type, for example:

`nand` is a function that accepts 2 `Bool` values, and checks if it at least one of them is `False`. We will show two ways to write this function.

The first is too look at the possible inputs and define the outputs directly:

```
nand :: Bool → Bool → Bool
nand False _     = True
nand True True   = False
nand True False = True
```

The other way is to define the function in terms of other functions and how the elements of the type `Bool` behave

```
nand :: Bool → Bool → Bool
nand a b = not (a && b)
```

The situation is something similar, for a lot of other types, like `Int`, `Char` and so on.

But with the addition of the List type from the previous chapter, we were able to add *new* information to a type, in the following sense:

Consider the type `[Int]`, the elements of these types are lists of integers, the way one would interact with these would be to treat it as a collection of objects, in which each element is an integer.

- so to write a function for this type, one first needs to think about the fact that the *shape* of an element looks like a list, and how one gets to the items of the list, and then treat the items like integers and write functions on them.
- A function for lists would thus have 2 components, at least conceptually if not explicit in the code itself, consider the following example:

  λ squaring all elements of a list

  ```
  squareAll :: [Int] → [Int]
  squareAll []       = []
  squareAll (x : xs) = x * x : squareAll xs
  ```

  Here, in the definition when we match patterns, we figure out the shape of the list element, and if we can extract an integer from it, then we square it and put it back in the list.

Something similar can be done with the type `[Bool]`:

- Once again, to write a function, one needs to first look at the *shape* an element as a list, Then pick elements out of them and treat them as `Bool` elements.
- An example of this will be the `and` function, that takes in a collection of `Bool` and returns `True` if and only if all of them are `True`.

λ and
```haskell
and :: [Bool] → Bool
and []       = True  -- We call scenarios like this 'vacuously true'
and (x : xs) = x && and xs
```

Once again, the pattern matching handles the shape of an element as a list, and the definition handles each item of a list as a `Bool`.

Then we see functions like the following:

- `drop`, which takes a list and discards the first couple of elements as specified.
- `elem`, which checks in an element belong to a list.
- `(==)`, which checks if 2 elements are equal.

Up until now, we had been emphasizing on the *shape* of elemets of a type, but these functions don't seem to care about it that much:

- The `drop` function just cares about the list structre of an element, and not what the internal item looks like.
- The `elem` function also does not care about the internal type of list items as long as some notion of equality if defined.
- The `(==)` works on all types where some notion of equality is defined (A counter example would be the type of functions: `Int → Int`).

Now one can define such functions for every single type, like:

```haskell
dropIntegers :: Int → [Integer] → [Integer]
dropIntegers = ...
dropChars :: Int → [Char] → [Char]
dropChars = ...
dropBools :: Int → [Bool] → [Bool]
dropBools = ...
.
.
.
```

but that has 2 problems:

- The first is that the defintion of all of these functions is the exact same, so doing this would be a lot of manual work, and one would also need to have different name for different types, which is very inconvenient.
- The second, and arguably a more serious issue, is that it stops us from abstracting, abstraction is the process of looking at a scenario and removing information that is not relevant to the problem.
  - An example would be that the `drop` simply lets us treat elememts as lists, while we can ignore the type of items in the list.
  - All of Mathematics and Computer Science is done like this, in some sense it is just that.
    - Linear Algebra lets us treat any set where addition and scaling is defined as one *kind* of thing, without worrying about any other structure on the elements.
    - Metric Spaces let us talk about all sets where there is a notion of distance.
    - Groups let us talk about sets where there is a notion of "combining" things together with more restriction.

in all of these fields of study, say linear algebra, a theorem generally involes working with an object, whose exact details we don't assume, just that it satisfies the conditions required for it to be a vector space and seeing what can be done with just that much information.

- And this is a powerful tool because solving a problem in the *abstract* version solves the problem in all *concretized* scenarios.

John Locke, An Essay Concerning Human Understanding (1690)

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called **abstraction**, and thus all its general ideas are made.

One of the ways abstraction is handled in Haskell, and a lot of other programming languages is **Polymorphism**.

Polymorphism

A **polymorphic** function is one whose output type depends on the input type. Such a property of a function is called **polymorphism**, and the word itself is ancient greek for ***many forms***.

A polymorphic function differs from functions we have seen in the following ways:

- It can take input from multiple differnt input types (not necessarily type, restrictions are allowed).
- Its output type can be differnt for different inputs types.

An example for such a function that we have seen in the previous section would be:

`drop`

```
drop :: Int → [a] → [a]
drop _ []     = []
drop 0 ls     = ls
drop n (x:xs) = drop (n-1) xs
```

The polymorphism of this function is shown in the type `drop :: Int → [a] → [a]` where we have used variables (usually called a type variable) instead of explicitly mentioning a type. This still has a lot of structure, and is not the same as forgetting about types, since, for instance, the same variable is used in both the second argument and the output, so they need to be of the same type, dropping some elements from a list of integers also gives a list of integers, we still have all the safety and correctness guarantees that types give us.

Datatypes of some list functions

A nice exercise would be to write the types of the following functions defined in the previous section: `head`, `tail`, `(!!)`, `take` and `splitAt`.

A Taste of Type Classes

Consider the case of the integer functions

```
f :: Int → Int
f x = x^2 + 2*x + 1

g :: Int → Int
g x = (x + 1)^2
```

We know that both functions, do the same thing in the mathematical sense, given any input, both of then have the same output, this is called function extensionality. But the does the following expression make sense in haskell?

λ **Function Extensionality**

```
f == g
```

This definitely seems like a fair thing to ask, as we already have a definition for equality of mathematical functions, but we run into 2 issues:

- Is it really fair to say that? In computer science, the way things are computed matter, that is where the subject gets its name from. A lot of times, one will be able to distinguish distinguish between functions, by simply looking at which one works faster or slower on big inputs, and that might be something people might want to factor into what they mean by "sameness". So maybe the assumption that 2 functions being equal pointwise imply the functions are equal may not be wise.
- The second is that in general it is not possible, in this case we have a mathematical identity that lets us prove so, but given any 2 function, it might be that the only way to prove that they are equal would be to actually check on every single value, and since domains of functions can be infinite, this would simply not be possible to compute.

So we can't have the type of `(==)` to be `a → a → Bool`. In fact, if I try to write it, the haskell compiler will complain to me by saying

```
funext.hs:8:7: error: [GHC-39999]
    • No instance for ‘Eq (Int → Int)’ arising from a use of ‘==’
      ... more error
  |
8 | h = f == g
  |
```

To tackle this, we define the following:

≒ **Typeclasses**

Typeclasses are a collection of types, characterized by the common ***behaviour***.

The previous section describes how one writes functions based on the *shape* of the objects. And that different types can have some aspects of their *shape* in common. λ **Function Extensionality** tells us that there are other properties shared by elements of different types, which we call their *behaviour*. By that we mean the functions that are defined for them.

Typeclasses are how one expresses in haskell, what a collection of types looks like, and the way to do so is by defining the common functions that work for all of them. Some examples are:

- `Eq`, which is the collection of all types for which the function `(==)` is defined.
- `Ord`, which is the collection of all types for which the function `(<)` is defined.
- `Show`, which is the collection of all types for which there is a function that converts them to `String` using the function `show`.

Note that in the above cases, defining one function lets you define some other functions, like `(≠)` for `Eq` and `(⩽)`, `(⩾)` and others for the `Ord` typeclass.

Now we come back to the `elem` function, the goal of this function is to check if a given element belongs to a list. And the following is a way to write it:

```
elem _ []       = False
elem e (x : xs) = e == x || elem e xs
```

Now lets try to give this a type.

First we see that the `e` must have the same types as the items in the list, but if we try to give it the type

```
elem :: a → [a] → Bool
```

we will encounter the same issue as we did in λ **Function Extensionality**, because of `(==)`. We need to find a way to say that `a` belongs to the collection `Eq`, and this leads to the correct type:

```
elem :: Eq a ⇒ a → [a] → Bool
```

X Checking if a list is sorted

Write the function `isSorted` which takes in a list as an argument, such that the elements of the list have a notion of ordering between them, and the output should be true if the list in an ascending order (equal elements are allowed to be next to each other), and false otherwise.

Higher Order Functions

One of the most powerful feature of functional programming languages is that it lets one pass in functions as argument to another function, and have funtions return other functions as outputs, these kinds of functions are known as:

≒ Higher Order Functions

A **higher order function** is a function that does at least one of the following things:

- It takes one or more functions as its arguments.
- It returns a function as an argument.

This is again a way of generalization and is very handy, as we will see in the rest of the chapter.

Currying

Perhaps the first place where we have encountered higher order functions is when we defined `(+) :: Int → Int → Int` way back in Chapter 3. We have been suggesting to think of the type as `(+) :: (Int, Int) → Int`, because that is really what we want the function to do, but in haskell it would actually mean `(+) :: Int → (Int → Int)`, which says the function has 1 interger argument, and it returns a function of type `Int → Int`.

An example from mathematics would be finding the derivative of a differentiable function f at a point x. This is generally represented as $f'(x)$ and the process of computing the derivative can be given to have the type

$$(f, x) \mapsto f'(x) : \left((\mathbb{R} \to \mathbb{R})^d \times \mathbb{R}\right) \to \mathbb{R}$$

Here $(\mathbb{R} \to \mathbb{R})^d$ is the type of real differentiable functions.

But one can also think of the derivative operator, that takes a differentiable function f and produces the function f', which can be given the following type:

$$\frac{d}{dx} : (\mathbb{R} \to \mathbb{R})^d \to (\mathbb{R} \to \mathbb{R})$$

In general, we have the following theorem:

Theorem **Currying**: Given any sets A, B, C, there is a *bijection* called curry between the sets $C^{A\times B}$ and the set $\left(C^B\right)^A$ such that given any function $f : C^{A\times B}$ we have

$$(\text{curry } f)(a)(b) = f(a, b)$$

Category theorists call the above condition *naturality* (or say that the bijection is *natural*). The notation Y^X is the set of functions from X to Y.

Proof We prove the above by defining $\text{curry} : C^{A\times B} \to \left(C^B\right)^A$, and then defining its inverse.

$$\text{curry}(f) :\equiv x \mapsto (y \mapsto f(x, y))$$

The inverse of curry is called uncurry : $\left(C^B\right)^A \to C^{A\times B}$

$$\text{uncurry}(g) :\equiv (x, y) \mapsto g(x)(y)$$

To complete the proof we need to show that the above functions are inverses.

Exercise

Show that the uncurry is the inverse of curry, and that the *naturality* condition holds.

(Note that one needs to show that uncurry is the 2-way inverse of curry, i.e, uncurry $\circ$ curry $=$ id and curry $\circ$ uncurry $=$ id, one direction is not enough.)

■

The above theorem, is a concretization of the very intuitive idea:

This may seem odd at first, but the relation between the two kinds of functions is not that hard to see, at least intuitively:

- Given a function f that takes in a pair of type $(A, B) \to C$, if one fixes the first argument, then we get a function $f(A, -)$ which would take an element of type B and then give an element of types C.
- But every different value of type A that we fix, we get a differnt function.
- Thus we can think of f as a function that takes in an element of type A and returns a function of type $B \to C$

And the above theorem is also "implemented" in haskell using the following functions:

curry and uncurry

```haskell
curry :: ((a, b) → c) → a → b → c
curry f a b = f (a, b)

uncurry :: (a → b → c) → (a, b) → c
uncurry g (a, b) = g a b
```

Currying lets us take a function with with argument, and lets us apply the function to each of them one at a time, rather than applying it on the entire tuple at once. One very interesting result of that is called **partial application**.

Partial applicaion is precisely the process of fixing some arugments to get a function over the remaining, let us look at some examples

```haskell
suc :: Int → Int
suc = (+ 1) -- suc 5 = 6

-- | curry examples
neg :: Int → Int
neg = (-1 *) -- neg 5 = -5
```

We will find many more examples in the next section.

Functions on Functions

We have already seen examples of a couple of functions whose arguments themselves are functions. The most recent ones being λ **curry and uncurry**, both of them take functions as inputs and return functions as outputs (note that our definition takes in functions and values, but we can always use partial application), these functions can be thought of as useful operations on functions.

Another very useful example, that a lot of us have seen is composition of functions, when we allow functions as inputs, composition can be treated like a function:

λ **composition**

```haskell
(.) :: (b → c) → (a → b) → (a → c)
g . f = \a → g (f a)

-- example
square :: Int → Int
square x = x * x

-- checks if a number is the same if written in reverse
is_palindrome :: Int → Bool
is_palindrome x = (s == reverse s)
  where
    s = show x -- convert x to string

is_square_palindrome :: Int → Bool
is_square_palindrome = is_palindrome . square
```

Breaking a complicated function into simpler parts, and being able to combime them is fair standard problem solving strategy, in both Mathematics and Computer Science, and in fact in a lot more general scenarios too! Having a clean notation for a tool that used fairly frequently is always a good idea!

Higher order functions are where polymorphism shines it brightest, see how the composition function works on all pairs of functions that can be composed in the mathematical sense, this would have been significantly less impressive if say it was only composition between functions from `Int → Int` and `Int → Bool`.

Another similar function that makes writing code in haskell much cleaner is the following:

λ **function application function**

```haskell
($) :: (a → b) → a → b
f $ a = f a
```

This may seem like a fairly trivial function that really doesn't offer anything apart from an extra `$`, but the following 2 lines make it useful

λ operator precedence

```
-- The 'r' in infixr says a.b.c = a.(b.c)
infixr 9 .
infixr 0 $
```

These 2 lines are saying that, whenever there is an expression, which contains both `($)` and `(.)`, haskell will first evaluate `(.)`, using these 2 one can write a chain of function applications as follows:

```
-- old way
f (g (h (i x)))

-- new way
f . g . h . i $ x
```

which in my opinion is much simpler to read!

Exercise

Write a function `repeat` that takes a function `f` and an argument `a` along with a natural number `n` and applies the function `n` times on `a`, for exaxmple: `repeat (+1) 5 3` would return `8`. Also figure out the type of the function.

A Short Note on Type Inference

Haskell is a statically typed language. What that means is that it requires the types for the data that is being processed by the program, and it needs to for an analysis that happens before running the program, this is called **type checking**.

It is not however required to give types to all functions (we do strongly recommend it though!), in fact one can simply not give any types at all. This is possible because the haskell compiler is smart enough to figure all of it out on its own! It's so good that when you do write type annotations for functions, haskell ignores it, figures the types out on its own and can then check if you have given the types correctly. This is called **type inference**.

Haskell's type inference also gives the most general possible type for a function. To see that, one can open ghci, and use the `:t` command to ask haskell for types of any given expression.

```
>>> :t flip
flip :: (a → b → c) → b → a → c
>>> :t (\ x y → x == y)
(\ x y → x == y) :: Eq a ⇒ a → a → Bool
```

The reader should now be equipped with everything they need to understand how types can be read and can now use type inference like this to understand haskell programs better.

Higher Order Functions on Maybe Type

The **Maybe Type**, as defined in Chapter 3 is another playground for higher order functions.

When using the `Maybe` types, one eventually runs into a problem that looks something like this:

- Break up the problem into a bunch of tiny steps, so make a lot of simple function and the final solution is to be achieved by combining all of them.

- Turns out that one the functions, maybe something in the very beginning returns a `Maybe Int` instead of an `Int`.
- This means that the next function along the chain, would have had to have its input type as `Maybe Int` to account for the potentially case of `Nothing`.
- This also forces the output type to be a `Maybe` type, this makes sense, if the process fails in the beginning, one might not want to continue.
- The `Maybe` now propogates in this manner through a large section of your code, this means that a huge chunk of code needs to be rewritten to looks something like:

```
f :: a → b
f inp = <some expression to produce output>

f' :: Maybe a → Maybe b
f' (Just inp) = Just $ <some epression to produce output>
f' Nothing    = Nothing
```

Note that `$` here is making our code a little bit cleaner, otherwise we would have to put the enter expression in paranthesis.

This is still not a very elegant way to write things though, and its just a lot of repetitive work (all of it is just book keeping really, one isn't really adding much to the program by making these changes, except for safety, programmers usually like to call it boilerplate.)

Instead of going and modifying each function manually, we make a function modifier, which is precisely what a higher order function: Our goal, which is obvious from the problem: `(a → b) → (Maybe a → Maybe b)` and we define it as follows:

λ maybeMap

```
maybeMap :: (a → b) → Maybe a → Maybe b
maybeMap f (Just a) = Just . f $ a
maybeMap _ Nothing  = Nothing

(<$>) :: (a → b) → Maybe a → Maybe b
f <$> a = maybeMap f a

(<.>) :: (b → c) → (a → Maybe b) → a → Maybe c
g <.> f = \x → g <$> f x

infixr 1 <$>
infixr 8 <.>
```

Note: The symbol `<$>` is written as `<$>`.

So consider the following chain of functions:

```
f . g . h . i . j $ x
```

where say `i` was the function that turned out to be the one with `Maybe` output, the only change we need to the code would be the following!

```
f . g . h <.> i . j $ x
```

X Beyond map

The above shows how haskell can elegantly handle cases when we want to convert a function from type `a → b` to a function from type `Maybe a → Maybe b`. This can be thought of as some sort of a *change in context*, where our function is now aware that its inputs can contain a possible fail value, which is `Nothing`. The reason for needing such a *change in context* were function of type `f :: a → Maybe b`, that is ones which can fail. They add the possiblility of failure to the *context*.

But since we have the power to be able to change *contexts* whenever wanted easily, we have a responsibility to keep it consistent when it makes sense. That is, what if there are multiple function with type `f :: a → Maybe b` we then would just want to use `<.>` or `maybeMap` to get something like:

```
v :: Maybe a
f :: a → Maybe b

g = f <$> a → Maybe (Maybe b)
```

This is most likely going to be undesirable, the point of `Maybe` was to say that there is a possiblility of error, the point of `maybeMap` was to propogate that possible error, so when there are multiple places where the program can fail, one can define `maybeJoin :: Maybe (Maybe a) → Maybe a`, with that we can have

```
g = maybeJoin $ f <$> a
```

This particular combination of doing `<$>` then `maybeJoin` will be very common, so people that use haskell put the 2 together in the function `(>>=) :: Maybe a → (a → Maybe b) → Maybe b` (the order of operands is reversed), this makes writing code so much cleaner, for instance:

```
val :: a
func1 :: a → Maybe b
func2 :: b → Maybe c
func3 :: c → Maybe d

final :: Maybe d
final = Maybe val
    >>= func1
    >>= func2
    >>= func3
```

Define `maybeJoin` and `(>>=)` and see how both of then are used in programs, and maybe compare then by how one would define `final` without these.

Note The symbol `(>>=)` is written as `(>>=)`.

Higher order functions, along with polymorphism help our code be really expressive, so we can write very small amounds of code that looks easy to read, which also does a lot. In the next chapter we will see a lot more examples of such functions.

Advanced List Operations

Arjun Maneesh Agarwal

advanced lists (feel free to change it)

List Comprehensions

As we have talked about before, Haskell tries to make it's syntax look as similer as possible to math notation. This is reprasented in one of the most powerful syntactic sugers in Haskell, list comprehension.

If we want to talk about all pythogorean triplets using integers from $1 - n$, we could express it mathematically as

$$\{(x, y, z) \mid x, y, z \in \{1, 2, ..., n\}, x^2 + y^2 = z^2\}$$

which can be written in Haskell as

```
[(x,y,z) | x ← [1..n], y ← [1..n], z ← [1..n], x^2 + y^2 == z^2]
```

This allows us to define a lof of operations we have seen before, in ch 1, in rather concise manner.

For example, `map :: (a→ b) → [a] → [b]` which used to apply a function to a list of elements of a suitable input type and gave a list of the suitable output type. Basically, `map f [a1,a2,a3] = [f a1, f a2, f a3]`. We can define this in two ways:

```
map _ [] = []
map f (x:xs) = (f x) : (map f xs)

-- and much more clearly and concisely as
map f ls = [f l | l ← ls]
```

Similerly, we had seen `filter :: (a → Bool) → [a] → [a]` which used to take a boolean function, some predicate to satisfy, and return the list of elements satisfying this predicate. We can define this as:

```
filter _ [] = []
filter p (x:xs) = let rest = p xs in
  if p x then x : rest else rest

-- and much more cleanly as
filter p ls = [l | l ← ls, p l]
```

Another operation we can consider, though not explictly defined in Haskell, is cartisian product. Hopefully, you can see where we are going with this right?

```
cart :: [a] → [b] → [(a,b)]
cart xs ys = [(x,y) | x ← xs, y ← ys]
```

Trying to define this reccursivly is much more cumbersome.

```
cart [] _ = []
cart (x:xs) ys = (go x ys) ++ (cart xs ys) where
 go _ [] = []
 go l (m:ms) = (l,m) : (go l ms)
```

Finally, let's talk a bit more about our pythogorean triplets example at the start of this section.

```
pythNaive :: Int → [(Int, Int, Int)]
pythNaive n = [(x,y,z) |
          x ← [1..n],
          y ← [1..n],
          z ← [1..n],
          x^2 + y^2 == z^2]
```

For `n = 1000`, we get the answer is some 13 minutes, which makes sense as our code is basically considering the 1000^3 triplets and then culling the ones which are not pythogorean. But could we do better?

A simple idea would be to not check for `z` as it is implied by the choice of `x` and `y` and instead set the condition as

```
pythMid n = [(x, y, z) |
    x ← [1..n],
    y ← [1..n],
    let z2 = x^2 + y^2,
    let z = floor (sqrt (fromIntegral z2)),
    z * z == z2]
```

This is clearly better as we will be only considering some 1000^2 triplets. Continuing with our example, for `n = 1000`, we finish in 1.32 seconds. As we expected, that is already much, much better than the previous case.

Also notice that we can define variables inside the comprehension by using the `let` syntax.

However, there is one final optimization we can do. The idea is that $x > y$ or $x < y$ for pythogorean triplets as `sqrt 2` is irrational. So if we can somehow, only evaluate only the cases where $x < y$ and then just genrate (x, y, z) and (y, x, z); we almost half the number of cases we check. This means, our final optimized code would look like:

```
pythOpt n = [t|
    x ← [1..n],
    y ← [(x+1)..n],
    let z2 = x^2 + y^2,
    let z = floor (sqrt (fromIntegral z2)),
    z * z == z2,
    t ← [(x,y,z), (y,x,z)]
    ]
```

This should only make some $\frac{1000*999}{2}$ triplets and cull the list from there. This makes it about twice as fast, which we can see as for $n = 1000$, we finish in 0.68 seconds.

Notice, we can't return two things in a list comprehension. That is, `pythOpt n = [(x,y,z), (y,x,z) | <blah blah>]` will given an error. Intead, we have to use `pythOpt n = [ t | <blah blah>, t ← [(x,y,z), (y,x,z)]]`.

Another intresting thing we can do using list comprehension is sorting. While further sorting methods and their speed is discussed in chapter 10, we will focus on a two methods of sorting: Merge Sort and Quick Sort.

We have seen the idea of divide and conquor before. If we can divide the problem in smaller parts and combine them, without wasting too much time in the spliting or combining, we can solve the problem. Both these methods work on this idea.

Merge Sort divides the list in two parts, sorts them and then merges these sorted lists by comparing element to elemeent. We can do this recursion with peace of mind as once we reach 1 element lists, we just say they are sorted. That is `mergeSort [x] = [x]`.

Just to illustrate, the merging would work as follows: `merge [1,2,6] [3,4,5]` would take the smaller of the two heads till both lists are empty. This works as as both the lists are sorted. The complete evaluation is something like:

```
merge [1,2,6] [3,4,5]
= 1 : merge [2,6] [3,4,5]
= 1 : 2 : merge [6] [3,4,5]
= 1 : 2 : 3 : merge [6] [4,5]
= 1 : 2 : 3 : 4 : merge [6] [5]
= 1 : 2 : 3 : 4 : 5 : merge [6] []
= 1 : 2 : 3 : 4 : 5 : 6 : merge [] []
= 1 : 2 : 3 : 4 : 5 : 6 : []
= [1,2,3,4,5,6]
```

So we can implement `merge`, rather simply as

```
merge :: Ord a ⇒ [a] → [a] → [a]
merge [] [] = []
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x < y
  then x : merge xs (y:ys)
  else y : merge (x:xs) ys
```

Note, we can only sort a list which has some definition of order on the elements. That is the elements must be of the typeclass `Ord`.

To implement merge sort, we now only need a way to split the list in half. This is rather easy, we have already seen `drop` and `take`. An inbuilt function in Haskell is `splitAt :: Int → [a] → ([a], [a])` which is basically equivalent to `splitAt n xs = (take n xs, drop n xs)`.

That means, we can now merge sort using the function

```
mergeSort :: Ord a ⇒ [a] → [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort left) (mergeSort right) where
  (left,right) = splitAt (length xs `div` 2) xs
```

X MergeSort Works?

Prove that merge sort indeed works. A road map is given

(i) Prove that `merge` defined by taking the smaller of the heads of the lists reccursivly, produces a sorted list given the two input lists were sorted. The idea is that the first element choosen has to be the smallest. Use induction of the sum of lengths of the lists.

(ii) Prove that `mergeSort` works using induction on the size of list to be sorted.

This is very efficent way to sort a list. If we define a function T count the number of operations we make, $T(n) < 2 * T(\lceil \frac{n}{2} \rceil) + n + n + n = 2 * T(\lceil \frac{n}{2} \rceil) + 3n$ where the 3 n's come from the length, the `splitAt` and the `merge`.

This implies

$$\begin{aligned}
T(n) &< n\lceil \log(n) \rceil T(1) + 3n + 3\left\lceil \frac{n}{2} \right\rceil + 3\left\lceil \frac{\lceil \frac{n}{2} \rceil}{2} \right\rceil + \ldots + 3 \\
&= n\lceil \log(n) \rceil + 3\left(n + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{\lceil \frac{n}{2} \rceil}{2} \right\rceil + \ldots + 1 \right) \\
&< n\lceil \log(n) \rceil + 3\left(n + \frac{n+1}{2} + \left\lceil \frac{n+1}{4} \right\rceil + \ldots + 1 \right) \\
&= n\lceil \log(n) \rceil + 3\left(n + \frac{n}{2} + \frac{1}{2} + \frac{n}{4} + \frac{1}{2} + \ldots + 1 \right) \\
&< n\lceil \log(n) \rceil + 3\left(2n + \frac{1}{2}\log(n) \right) \\
&= n\lceil \log(n) \rceil + 6n + \frac{3}{2}\log(n) \\
&< n(\log(n) + 1) + 6n + \frac{3}{2}\log(n) \\
&< n\log(n) + 7n + \frac{3}{2}\log(n)
\end{aligned}$$

Two things to note are that the above computation was very cumbersome. We will later see a way to make it a bit less cumbersome, at the cost of some information.

The second, for sufficiently large n, $n\log(n)$ dominates the equation. That is

$$\exists m \; s.t. \; \forall n > m : n\log(n) > 6n > \frac{3}{2}\log(n)$$

This means that as n becomes large, we can sort of ignore the other terms. We will later prove, that given no more information other than the fact that the shape of the elemeents in the list is such that they can be compared, we can't do much better. The dominating term will be $n\log(n)$ times some constant. This later refers to chapter 10.

In this algorithm, we waste some ammount of operations dividing the list in 2. What if we take our odds and approximatly divide the list into two parts?

This is the idea of quick sort. If we take a random element in the list, we expect half the elements to be lesser than it and half to be greater. We can use this fact to define quickSort by splitting the list on the basis of the first element and keep going. This can be implemented as:

```
quickSort :: Ord a ⇒ [a] → [a]
quickSort [] = []
quickSort [x] = [x]
quickSort (x:xs) = quickSort [l | l ← xs, l > x] ++ [x] ++ quickSort [r |
r ← xs, r ⩽ x]
```

This is not the most efficent way to do so as we go through the list twice, but it is the most aesthetically pleasing and concise.

X Faster Quick Sort

A slight improvment can be made to the implementation by not using list comprehension and instead using a helper function, to traverse the list only once.

Try to figure out this implementation.

We will use the above deined version for our calculations, although you are reccomended to re-do the with the more efficient implementation as well.

X Quick Sort works?

Prove that Quick Sort does indeed works. This has to be done by

Clearly, With n being the length of list, $T(n)$ is a random variable dependent on the permutation of the list.

Let l be the number of elements less than the first elements and $r = n - l - 1$. This means $T(n) = T(l) + T(r) + 2(n-1) + l + 1$ where the $n-1$ comes from the list comprehension and the $l+1$ from the concatination.

In the worst case scenario, our algoritm could keep spliting the list into a length 0 and a length $n-1$ list. This would screw us very badly. Furthermore, as this is the worst case, our algorithm also gets the worst concatination, that is the left list is $n-1$.

As $T(n) = T(0) + T(n-1) + 2(n-1) + (n-1) + 1$ where the $n-1$ comes from the list comprehension and the $(n-1)+1$ from the concatination. Using $T(0) = 1$, This evaluates to

$$\begin{aligned} T(n) &= T(0) + T(n-1) + 2(n-1) + (n-1) + 1 \\ &= T(n-1) + 3n - 1 \\ &= 3(n + (n-1) + \ldots + 1)\underbrace{-1 - 1 \ldots - 1}_{n \text{ times}} \\ &= 3\frac{n(n+1)}{2} - n \\ &= \frac{3}{2}n^2 + \frac{1}{2}n \end{aligned}$$

Which is quite bad. Furthermore, while this case is based on already sorted list; we don't fare better in the case where the list split into $n-1$ and 0 every stage despite someconcatinaions costing 1.

The above case is also common enough. How common?

X A Strange Proof

Prove $2^{n-1} \leq n!$

Then why are we intrested in Quick Sort? and why is named quick?

Let's look at the average or expected time it would take.

$$\begin{aligned}\mathbb{E}(T(n)) &= \mathbb{E}(T(l) + T(r) + 2(n-1) + l + 1) \\ &= \mathbb{E}(T(l)) + \mathbb{E}(T(r)) + 2(n-1) + 1 + \mathbb{E}(l) \\ &\text{Using the fact } l \text{ and } r \text{ are symmetrical and interchangable and assuming } l \text{ is uniform.} \\ &= 2\mathbb{E}(T(l)) + 2(n-1) + 1 + \frac{n-1}{2} \\ &= \frac{2}{n}\sum_{i=0}^{n-1}\mathbb{E}(T(i)) + \frac{5}{2}n - \frac{3}{2}\end{aligned}$$

Now for the sake of simplification, we will take $n\mathbb{E}(T(n)) = S(n)$.

$$\begin{aligned}S(n) &= 2\sum_{i=0}^{n-1}\mathbb{E}(T(i)) + \frac{5}{2}n^2 - \frac{3}{2}n \\ &\text{and } S(n-1) = 2\sum_{i=0}^{n-2}\mathbb{E}(T(i)) + \frac{5}{2}(n-1)^2 - \frac{3}{2}(n-1) \\ \Rightarrow S(n) - S(n-1) &= 2\mathbb{E}(T(n-1)) + \frac{5}{2}(2n-1) - \frac{3}{2} \\ \Rightarrow n\mathbb{E}(T(n)) - (n-1)\mathbb{E}(T(n-1)) &= 2\mathbb{E}(T(n-1)) + 5n - 4 \\ \Rightarrow \mathbb{E}(T(n)) &= \frac{n+1}{n}\mathbb{E}(T(n-1)) + 5 - \frac{4}{n}\end{aligned}$$

This reccurence can actually be solved! Making the magical substitution $f(n) = \frac{\mathbb{E}(T(n))}{n+1}$

$$\begin{aligned}(n+1)f(n) &= (n+1)f(n-1) + 5 - \frac{4}{n} \\ \Rightarrow f(n) &= f(n-1) + \frac{5n-4}{n(n+1)} \\ \Rightarrow f(n) &= f(1) + \sum_{i=2}^{n}\frac{5i-4}{i(i+1)} \\ \Rightarrow f(n) &= 1 + \sum_{i=2}^{n}\frac{5i-4}{i(i+1)}\end{aligned}$$

Finally, this is something we can work with. Using the fraction decomposition technique, We can write $\frac{5i-4}{i(i+1)} = \frac{9}{i+1} - \frac{4}{i}$

$$\begin{aligned}f(n) &= 1 + \sum_{i=2}^{n}\frac{9}{i+1} - \sum_{i=2}^{n}\frac{4}{i} \\ &= 1 + \frac{9}{n+1} + 5\sum_{i=3}^{n}\frac{1}{i} - 2 \\ &\approx \frac{9}{n+1} - 1 + 5\left(\log(n) + \gamma - 1 - \frac{1}{2}\right) \\ \Rightarrow \mathbb{E}(T(n)) &= (n+1)f(n) = 9 - (n+1) + 5(n+1)\left(\log(n) + \gamma - \frac{3}{2}\right) \\ &= 5n\log(n) - \left(5\gamma - \frac{17}{2}\right)n + 5\log(n) + 5\gamma - \frac{1}{2}\end{aligned}$$

Where $\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i}\right) - \log(n) = 0.577...$ is called the Euler–Mascheroni constant.

The point of this, frankly exhausting, calculation is that despite having an exponential number of cases in quadratic time (remember the worst case analysis?), Quick Sort's expected number of operations is still a constent times $n\log(n)$ which is optimal.

This implies that there are some lists where Quick Sort is extreamly efficent and as one might expect there are many such lists. This is why languages which can keep states (C++, C, Rust etc) etc use something called Introsort which uses Quick Sort till the depth of recusion reaches $\log(n)$ (at which point it is safe to say we are in one of the not nice cases); then we fallback to Merge Sort or a Heap Sort(which we will see in chapter 11).

Haskell used to use quickSort as the default but in 2002, Ian Lynagh changed it to Merge Sort. This was motivated by the fact that Merge Sort gurentees sorting in some time while Quick Sort will sometimes finish much quicker and other times, alas.

Zip it up!

A simple thing we might want to do at times is to join two lists together as pairs.

Introduction to Datatypes

Shubh Sharma

pre-complexity data types (feel free to change it)

- Define recurssion in recursive data types and define (4)
- define Nat, List, Tree

Computation as Reduction

Shubh Sharma

computation (feel free to change it)

Complexity

Arjun Maneesh Agarwal

complexity (feel free to change it)

Advanced Data Structures

Arjun Maneesh Agarwal

post-complexity data types (feel free to change it)

- Queue
- Segment Tree
- BST
- Set
- Map
- Define recurssion in recursive data types and define (4)
- define Nat, List, Tree

Type Classes

Ryan Hota

typeclasses (feel free to change it)

Monads

Ryan Hota

Monad (feel free to change it)