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 May 25, 2025.

To someone

Table of Contents

Basic Theory

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

functions (feel free to change it)

Haskell Setup on Linux

Shubh Sharma

setup

setup linux (feel free to change it)

Haskell Setup on MacOS

Arjun Maneesh Agarwal

setup mac (feel free to change it)

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[1] which is quite accurate

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

and 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.

Contents

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

```
(@@) :: 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

```
-- 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?

Contents

λ 17 Xors contd.

```haskell
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.

```haskell
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

Contents

λ 17 Xors, cotd

```haskell
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

```haskell
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

```
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.

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`.[2]

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

[2]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.

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
```

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 truncating 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

```haskell
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

Exponantion

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

What can we say about the three exponation operators?

```
<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
199506311688075838488374216268358508382 ...
>>> 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**

```haskell
-- 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 $p * q = \gcd(p, q) * \operatorname{lcm}(p, q)$. 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 slightltly 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

```haskell
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`).

Basic String Operations

We will now talk about string operations. As we mentioned in the start, strings are a list of characters. This autmoatically implies a lot of advanced string related operations would need to go through lists. Hence, we will only cover the basic ones here.

```
>>> "hello" ++ " " ++ "world!"
"hello world"
>>> 'h' : "ello world"
"hello world"
>>> 'h' ++ "ello world"
<interactive>:78:1: error: [GHC-83865]
>>> "hello" : " " : "world"
<interactive>:79:17: error: [GHC-83865]
>>> length "hello"
5
>>> reverse "hello"
"olleh"
>>> take 3 "hello"
"hel"
>>> drop 3 "hello"
"lo"
>>> splitAt 3 "hello"
("hel","lo")
```

We want you to observe and infer that `(++)` is used to concat two strings while `(:)` is used to append a character to a string. This distinction matters as doing it any other way creates a horrible error.

`length` provides the length of the string, `reverse` reverses the string and `take` and `drop` allow us to either take the first few elements of a string or dispose of them. One could simply define `splitAt n str = (take n str, drop n str)`.

Note, `(:)` is a primitive and is defined axiomatically. That is, we cannot breakdown it's implementation. It just exists and works. The exact working will become more clearer in chapter 9 and 11.

Defining the other functions is just an excercise in recursion, and a very straightforward one at that. The only one with any cleverness will be `reverse`, but alas we have already seen it in chapter 1.

```
(++) :: String → String → String
[] ++ str = str
(x:xs) ++ ys = x : (xs ++ ys)

length :: String → Int
length "" = 0
length (x:xs) = 1 + length xs

take :: Int → String → Int
take _ "" = ""
take 0 str = str
take n (x:xs) = x : (take (n-1) xs)

drop :: Int → String → String
drop _ "" = ""
drop 0 str = str
drop n (x:xs) = drop (n-1) xs

naiveReverse :: String → String
naiveReverse "" = ""
naiveReverse (x:xs) = (naiveReverse xs) ++ [x]

betterReverse :: String → String
betterReverse (x:xs) = go (x:xs) [] where
  go [] rev = rev
  go (x:xs) rev = go xs (x:rev)
```

`naiveReverse` clearly uses some $\frac{n(n+1)}{2}$ append operations where n is the length of the list as we use concat unnecesarily and it is expensive. On the other hand, `betterReverse` usea only some `n` append operations. This makes it much faster and is indeed how `reverse` is defined.

Dealing with Characters

We will now talk about characters. Haskell packs up all the functions relating to them in a module called `Data.Char`. We will explore some of the functions there.

So if you are following along, feel free to enter `import Data.Char` in your ghci or add it to the top of your haskell file.

The most basic and importent functions here are `ord` and `chr`. Characters, like the ones you are reading now, are represented inside a computer using numbers. These numbers are part of a standard called ASCII (American Standard Code for Information Interchange), or more generally, Unicode.

In Haskell, the function `ord` takes a character and returns its corresponding numeric code (called its code point). The function `chr` does the reverse: it takes a number and returns the character it represents.

```
>>> ord 'g'
103
>>> ord 'G'
71
>>> chr 71
'G'
>>> chr '103
'g'
```

The ASCII standard originally defined 128 characters, numbered from 0 to 127. These include English letters, digits, punctuation, and some control characters that do not represent symbols but serve technical purposes. For example,

- `'\SOH'` - start of heading
- `'\EOT'` - end of transmission(used in sending data to denote if sending is over. If it is not recived, an error may be percived.)
- `'\ETX'` - end of text
- `'\a'` - alert(sometimes denoted as `'\BEL'`)
- `\n` - new line(sometimes denored as `'\LF'`)
- `'\t'` - horizontal tab(sometimes denored as `'\HT'`)
- `'\SP'` - space(often denoted as `' '`)

While `chr :: Char -> Int` is defined on all valid characters, `ord :: Int -> Char` is not defined on all integers for the reason that we don't have as many characters as integers.

The first defined character, aka `ord 0`, is `\NUL`. It is a control character used to represents the null character or nothing. `ord (-1)` results in an error, as does any other negitive number.

We have in total 34 such control characters. From 0 — 32 and then at 127(`'\DEL'`, used to log the use of delete. Not to be confused with `'\BS'` which is used to log backspaces).

Although the ASCII range ends at 127, because it was designed for a 7-bit system, modern systems use Unicode, which extends this idea and assigns unique numbers to over a million characters - including symbols from nearly every language. As Haskell uses Unicode under the hood. That means ord and chr can go well beyond ASCII:

```
>>> ord '☃'
9731
>>> chr 9731
'☃'
```

We will not go into how `ord` and `chr` are implemented, as that involves lower-level details. Just know that they work reliably and are part of the Haskell standard library.

With this out of the way, we can look at some more char based functions.

```
>>> isLower 'a'
True
>>> isLower 'A'
False
>>> isLower ','
False
>>> isUpper 'a'
False
>>> isUpper 'A'
True
>>> isUpper ','
False

>>> toLower 'a'
'a'
>>> toLower 'A'
'a'
>>> toLower ','
,
>>> toUpper 'a'
'A'
>>> toUpper 'A'
'A'
>>> toUpper ','
,
```

Simmiler functions are `isSpace, isDigit, isAlpha, isAlphaNum` for white spaces(space, tab, newline), digits, alphabets and alphanumerics(alphabets or number).

This means one could simply turn a whole string lower case or filter out only the alphabnumeric characters using `map` and `filter`.

```
>>> map toLower "HelLo WOrlD, I am Mixed uP"
"hello world, i am mixed up"

>>> filter isAlphaNum "mix^@($ed &(u*(!p m&(!^#e)*!^ss"
"mixedupmess"
```

< I will complete this section later. >

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, square root and cube root are well behaved, implement Newton–Raphson method.

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,

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

≑ **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***.

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,

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

declaring a variable

Declare a variable of type `Char`.

`A → B` is analogous to B^A or 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 -

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

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

basic function definition

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

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)` -

type of a pair
```haskell
>>> :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 -

elements of a product type
```haskell
>>> 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 -

first component of a pair
```haskell
fst (a,b) = a
```

and a similar function to get the second component -

second component of a pair
```haskell
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` $\cup$ `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.

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 - $(\{extra\ element\} \sqcup$ `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 $(\{extra\ element\} \sqcup$ `Rational` $)$ as the type `Either () Rational`.

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

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 **reciprocal using either** using `Maybe` instead -

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`.

Contents

- 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

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 (feel free to change it)

Advanced List Operations

Shubh Sharma

advanced lists (feel free to change it)

Introduction to Datatypes

Arjun Maneesh Agarwal

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)