Haskell for CMI

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

Haskell for CMI

This is (still!) an incomplete draft.
Please send any corrections, comments etc. to feedback_host@mailthing.com
Last updated September 16, 2025.

To someone

Table of Contents

Basic Theory

§1.1. Precise Communication

Haskell (as well as a lot of other programming languages) and Mathematics, both involve communicating an idea in a language that is precise enough for them to be understood without ambiguity.

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 interprets it into steps for manipulating some expression, or doing some action.

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 the context of Haskell, computers, being machines, are extremely unimaginative, and do not possess any inherent pattern-sensing abilities. 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**.

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

  etc.

§1.3. 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.

The following examples should clarify further.

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 d , defined as the smallest number in the set $\{n \in \mathbb{N} \mid \exists \text{ infinitely many pairs } (p, q) \text{ of prime numbers with } |p - q| \leq n\}$

Therefore we can see that relations and functions can also be **values**, as long as they are specific and not scenario-dependent. For example, the order $<$ on $\mathbb{N}$ does not have different meanings or interpretations in different scenarios, but rather has a fixed meaning which is independent of whatever the context is.

In fact, as we see in the last example, we don't even currently know the exact value of d.
The famous "Twin Primes Conjecture" is just about whether $d == 2$ or not.

So, the moral of the story is that even if we don't know what the exact value is,
we can still know that it is **some** ≑ **mathematical value**,
as it does not change from scenario to scenario and remains constant, even though it is an unknown constant.

§1.4. Variables

≑ **mathematical variable**

A **mathematical variable** is a symbol or chain of symbols
meant to represent an arbitrary element from a set of ≑ **mathematical values**,
usually as a way to show that whatever process follows is general enough so that the process can be carried out with any arbitrary value from that set.

The following examples should clarify further.

For example, consider the following function definition -

$$f : \mathbb{R} \to \mathbb{R}$$
$$f(x) := 3x + x^2$$

Here, x is a ≑ **mathematical variable** as it isn't any one specific ≑ **mathematical value**,
but rather **represents an arbitrary** element from the set of real numbers.

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

$$n + 1 > n$$

Hence Proved !! ■

Here, n is a ≑ **mathematical variable** as it isn't any one specific ≑ **mathematical value**,
but rather **represents an arbitrary** element from the set of natural numbers.

Here is another theorem -

Theorem For any $f : \mathbb{N} \to \mathbb{N}$, if f is a strictly increasing function, then f(0) < f(1)

Proof Let $f : \mathbb{N} \to \mathbb{N}$ be a strictly increasing function. Thus

$$\forall n, m \in \mathbb{N}, n < m \Rightarrow f(n) < f(m)$$

Take n to be 0 and m to be 1. Thus we get

$$f(0) < f(1)$$

Hence Proved! ■

Here, f is a ≑ **mathematical variable** as it isn't any one specific ≑ **mathematical value**, but rather **represents an arbitrary** element from the set of all $\mathbb{N} \to \mathbb{N}$ strictly increasing functions.

It has been used to show a certain fact that holds for **any** natural number.

§1.5. Well-Formed Expressions

Consider the expression -

$$xyx \Longleftarrow \forall \Rightarrow f(\nleftrightarrow > \vec{v}$$

It is an expression as it **is** a bunch of symbols arranged one after the other, but the expression is obviously meaningless.

So what distinguishes a meaningless expression from a meaningful one? Wouldn't it be nice to have a systematic way to check whether an expression is meaningful or not?

Indeed, that is what the following definition tries to achieve - a systematic method to detect whether an expression is well-structured enough to possibly convey any meaning.

≑ **checking whether mathematical expression is well-formed**

It is difficult to give a direct definition of a **well-formed expression**.

So before giving the direct definition,
we define a ***formal procedure*** to check whether an expression is a **well-formed expression** or not.

The procedure is as follows -

Given an expression e,

- first check whether e is
  - a ≑ **mathematical value**, or
  - a ≑ **mathematical variable**

  in which cases e passes the check and is a **well-formed expression**.

Failing that,

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

And only if e passes this check will it be a **well-formed expression**.

≑ **well-formed mathematical expression**

A ***mathematical expression*** is said to be a **well-formed mathematical expression** if and only if it passes the formal checking procedure defined in ≑ **checking whether mathematical expression is well-formed**.

Let us use ≑ **checking whether mathematical expression is well-formed** 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 whether expression is well-formed**

Suppose a, b, c, v, f, g are ≑ **mathematical values**.
Suppose x, y, n, h are ≑ **mathematical variables**.

Check whether the expression

$$f(g(x, y), f(a, h(v), c), h(h(h(n))))$$

is well-formed or not.

§1.6. Function Definitions

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 function definitions in mathematics work**.

§1.6.1. Using Expressions

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

A few examples -
- $f(x) := x^3 \cdot x^5 + x^2 + 1$
- $\mathrm{second}(a, b) := b$
- $\zeta(s) := \sum_{n=1}^{\infty} \frac{1}{n^s}$

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.

§1.6.2. 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.

§1.6.2.1. Where, Let

Consider the definition of the famous sine function -

$$\mathrm{sine} : \mathbb{R} \to \mathbb{R}$$

Given an angle θ,
Let T be a right-angled triangle, one of whose angles is θ.
Let p be the length of the perpendicular of T.
Let h be the length of the hypotenuse of T.
Then

$$\mathrm{sine}(\theta) := \frac{p}{h}$$

Here we use the variables p and h in the right-hand side of the definition, but to get their meanings one will have to look at how they are defined beforehand in the lines beginning with “let”.

We can also do the exact same thing using “where” instead of “let”.

$$\begin{aligned} \mathrm{sine} &: \mathbb{R} \to \mathbb{R} \\ \mathrm{sine}(\theta) &:= \frac{p}{h} \\ &\text{,where} \\ &\qquad T := \text{a right-angled triangle with one angle} == \theta \\ &\qquad p := \text{the length of the perpendicular of } T \\ &\qquad h := \text{the length of the hypotenuse} \quad \text{of } T \end{aligned}$$

Here we use the variables p and h in the right-hand side of the definition, but to get their meanings one will have to look at how they are defined after “where”.

[1]In order to have a clear distinction between definition 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”.

§1.6.2.2. Anonymous Functions

A function definition such as

$$
\begin{aligned}
&f : \mathbb{R} \to \mathbb{R} \\
&f(x) \coloneqq x^3 \cdot x^5 + x^2 + 1
\end{aligned}
$$

for convenience, can be rewritten as -

$$(x \mapsto x^3 \cdot x^5 + x^2 + 1) : \mathbb{R} \to \mathbb{R}$$

Notice that we did not use the symbol f, which is the name of the function, which is why this style of definition is called “anonymous”.

Also, we used $\mapsto$ in place of $\coloneqq$

This style 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 -

$$
\begin{aligned}
&\mathrm{harmonicSum} : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \to \mathbb{R}_{>0} \\
&\mathrm{harmonicSum}(x, y) \coloneqq \frac{1}{x} + \frac{1}{y}
\end{aligned}
$$

which, for convenience, can be rewritten as -

$$\left(x, y \mapsto \frac{1}{x} + \frac{1}{y}\right) : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \to \mathbb{R}_{>0}$$

§1.6.2.3. Piecewise Functions

Sometimes, the expression on the right-hand side of the definition needs to depend upon some condition, and we denote that in the following way -

$$
< \mathrm{functionName} > (x) \coloneqq \begin{cases}
< \mathrm{expression}_1 > \ ; \text{ if } < \mathrm{condition}_1 > \\
< \mathrm{expression}_2 > \ ; \text{ if } < \mathrm{condition}_2 > \\
< \mathrm{expression}_3 > \ ; \text{ if } < \mathrm{condition}_3 > \\
. \\
. \\
. \\
< \mathrm{expression}_n > \ ; \text{ if } < \mathrm{condition}_n >
\end{cases}
$$

For example, consider the following definition -

$$
\begin{aligned}
&\mathrm{signum} : \mathbb{R} \to \mathbb{R} \\
&\mathrm{signum}(x) \coloneqq \begin{cases}
+1 \ ; \text{ if } x \ > \ 0 \\
\ \ 0 \ ; \text{ if } x == 0 \\
-1 \ ; \text{ if } x \ < \ 0
\end{cases}
\end{aligned}
$$

The “signum” of a real number tells the “sign” of the real number ; whether the number is positive, zero, or negative.

§1.6.2.4. Pattern Matching

Pattern Matching is another way to write piecewise definitions which can work in certain situations.

For example, consider the last definition -

$$\text{signum}(x) := \begin{cases} +1 \text{ ; if } x \ > \ 0 \\ \ \ 0 \text{ ; if } x == 0 \\ -1 \text{ ; if } x \ < \ 0 \end{cases}$$

which can be rewritten as -

$$\text{signum}(0) := 0$$
$$\text{signum}(x) := \frac{x}{|x|}$$

This definition relies on checking the form of the input.

If the input is of the form “0”, then the output is defined to be 0.
For any other number x, the output is defined to be $\frac{x}{|x|}$

However, there might remain some confusion -
If the input is “0”, then why can’t we take x to be 0, and apply the second line ($\text{signum}(x) := \frac{x}{|x|}$) of the definition ?

To avoid this confusion, we adopt the following convention -
Given any input, we start reading from the topmost line of the function definition to the bottom-most, and we apply the first applicable definition.

So here, the first line ($\text{signum}(0) := 0$) will be used as the definition when the input is 0.

§1.6.3. 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 -

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

§1.6.3.1. 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, $f(4)$.

Contents

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**
if and only if
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**.

§1.6.3.2. Induction

÷ **principle of mathematical induction**

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

- φ_0 is true
- For each $n > 0$, if φ_{n-1} is true, then φ_n is also true.

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

The above definition should be read as follows, given a sequence of formulas:

- The first one is true.
- Any formula being true, implies that the next one in the sequence is true.

Then all of the formulas in the sequence are true. Something like a chain of dominoes falling.

x **Exercise**

Show that n^2 is the same as the sum of first n odd numbers using induction.

X The scenic way

(a) Prove the following theorem of Nicomachus by induction:

$$
\begin{gathered}
1^3 = 1 \\
2^3 = 3 + 5 \\
3^3 = 7 + 9 + 11 \\
4^3 = 13 + 15 + 17 + 19 \\
\vdots
\end{gathered}
$$

(b) Use this result to prove the remarkable formula

$$1^3 + 2^3 + \dots + n^3 = (1 + 2 + \dots + n)^2$$

X There is enough information!

Given $a_0 = 100$ and $a_n = -a_{n-1} - a_{n-2}$, what is a_{2025}?

X 2-3 Color Theorem

A k-coloring is said to exist if the regions the plane is divided off in can be colored with three colors in such a way that no two regions sharing some length of border are the same color.

(a) A finite number of circles (possibly intersecting and touching) are drawn on a paper. Prove that a valid 2-coloring of the regions divided off by the circles exists.

(b) A circle and a chord of that circle are drawn in a plane. Then a second circle and chord of that circle are added. Repeating this process, until there are n circles with chords drawn, prove that a valid 3-coloring of the regions in the plane divided off by the circles and chords exists.

X Square-full

Call an integer square-full if each of its prime factors occurs to a second power (at least). Prove that there are infinitely many pairs of consecutive square-fulls.

Hint: We recommend using induction. Given $(a, a+1)$ are square-full, can we generate another?

X Same Height?

Here is a proof by induction that all people have the same height. We prove that for any positive integer n, any group of n people all have the same height. This is clearly true for $n = 1$. Now assume it for n, and suppose we have a group of $n + 1$ persons, say $P_1, P_2, \cdots, P_{n+1}$. By the induction hypothesis, the n people $P_1, P_2, \cdots, P_n$ all have the same height. Similarly the n people $P_2, P_3, \cdots, P_{n+1}$ all have the same height. Both groups of people contain $P_2, P_3, \cdots, P_n$, so P_1 and P_{n+1} have the same height as $P_2, P_3, \cdots, P_n$. Thus all of $P_1, P_2, \cdots, P_{n+1}$ have the same height. Hence by induction, for any n any group of n people have the same height. Letting n be the total number of people in the world, we conclude that all people have the same height. Is there a flaw in this argument?

✗ proving the principle of induction

Prove that the following statements are equivalent -
- every nonempty subset of $\mathbb{N}$ has a smallest element
- the ≑ **principle of mathematical induction**

You can assume that $<$ is a linear order on $\mathbb{N}$ with $n-1<n$ and such that there are no elements strictly between $n-1$ and n.

§1.6.3.3. 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 -

- **⟨⟨ φ_0 is true ⟩⟩**
  The only natural number which is ≤ 0 is 0, and $F(0) := 1$, so the definition terminates immediately.
- **⟨⟨ For each $n > 0$, if φ_{n-1} is true, then φ_n is also true. ⟩⟩**
  Assume that φ_{n-1} is true.
  Let m be an arbitrary natural number which is $\leq n$.
  - ⟨⟨ Case 1 $(m \leq 1)$ ⟩⟩
    $F(m) := 1$, so the definition terminates immediately.
  - ⟨⟨ Case 2 $(m > 1)$ ⟩⟩
    $F(m) := F(m-1) + F(m-2)$,
    and since $m-1$ and $m-2$ are both $\leq n-1$,
    φ_{n-1} 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 is proved!

Hence the theorem is proved!! ■

§1.7. Infix Binary Operators

Usually, the name of the function is written before the inputs given to it. For example, we can see that in the expression $f(x, y, z)$, the symbol f is written to the left of / before any of the inputs x, y or z.

However, it's not always like that. For example, take the expression

$$x + y$$

Here, the function name is $+$, and the inputs are x and y.

But $+$ has been written in-between x and y, not before!

Such a function is called an infix binary operator[2]

≑ **infix binary operator**

An **infix binary operator** is a ***function*** which takes exactly 2 inputs and whose function name is written between the 2 inputs rather than before them.

Examples include -
- $+$ (addition)
- $-$ (subtraction)
- $\times$ or $*$ (multiplication)
- $/$ (division)

§1.8. Trees

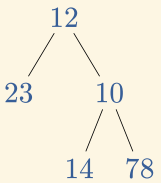
Trees are a way to structure a collection of objects.

Trees are a fundamental way to understand expressions and how haskell deals with them.

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

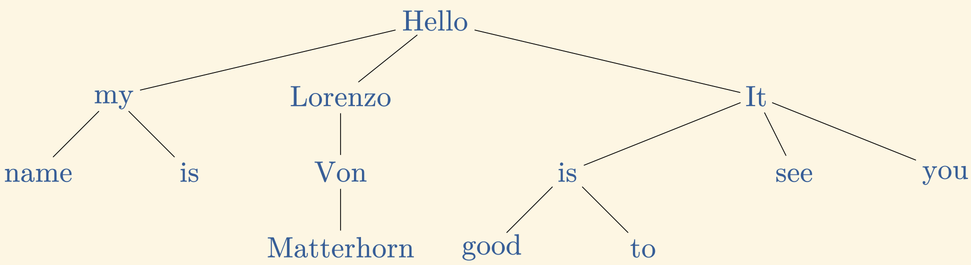
§1.8.1. Examples of Trees

Here we have a tree which defines a structure on a collection of natural numbers -



The line segments are what defines the structure.

The following tree defines a structure on a collection of words from the English language -



§1.8.2. Making Larger Trees from Smaller Trees

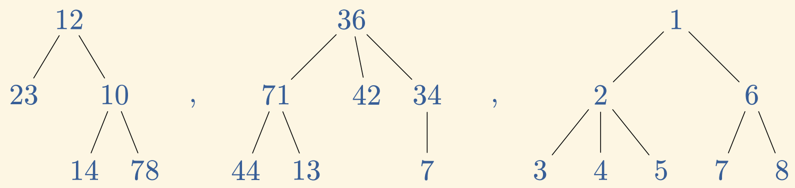
If we have an object -

$$89$$

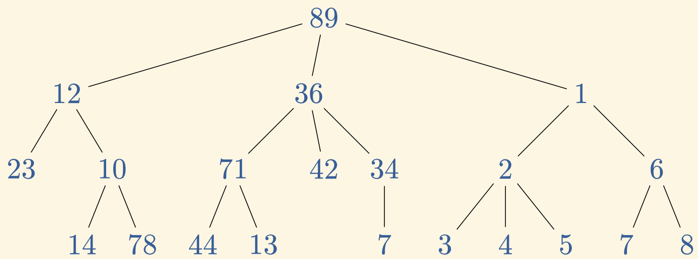
and a few trees -

[2]

infix - because the function name is **in-between** the inputs
binary - because exactly **2** inputs, and binary refers to **2**
operator - another way of saying **function**



we can put them together into one large tree by connecting them with line segments, like so -



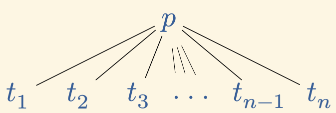
In general, if we have an object

$$p$$

and a bunch of trees

$$t_1, t_2, t_3, \ldots, t_{n-1}, t_n$$

, we can put them together in a larger tree, by connecting them with n line segments, like so -



We would like to define trees so that only those which are made in the above manner qualify as trees.

§1.8.3. Formal Definition of Trees

A **tree over a set** S defines a meaningful structure on a collection of elements from S.
The examples we've seen include trees over the set $\mathbb{N}$, as well as a tree over the set of English words.

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.

≑ **checking whether object is tree**

The formal 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 $\begin{array}{c} p \\ t_1 \quad t_2 \quad t_3 \quad \cdots \quad t_{n-1} \quad t_n \end{array}$, where
  - $p \in S$
  - and each of $t_1, t_2, t_3, \ldots, t_{n-1}$, and t_n is a **tree over S**.

≑ **tree**

Given a set S, a ***mathematical object*** is said to be a **tree over S** if and only if it passes the formal checking procedure defined in ≑ **checking whether object is tree**.

Let us use this definition to check whether

$$\begin{array}{ccc} & 12 & \\ / & & \backslash \\ 23 & & 10 \\ & / & \backslash \\ & 14 & 78 \end{array}$$

is a **tree over the natural numbers**.

Let's start -

$\begin{array}{ccc} & 12 & \\ / & & \backslash \\ 23 & & 10 \\ & / & \backslash \\ & 14 & 78 \end{array}$ is of the form $\begin{array}{ccc} & p & \\ / & & \backslash \\ t_1 & & t_2 \end{array}$, where p is 12 , t_1 is 23 , and t_2 is $\begin{array}{ccc} & 10 & \\ / & & \backslash \\ 14 & & 78 \end{array}$.

Of course, $12 \in \mathbb{N}$ and therefore $p \in S$.

So we are only left to check that 23 and $\begin{array}{ccc} & 10 & \\ / & & \backslash \\ 14 & & 78 \end{array}$ are trees over the natural numbers.

$23 \in \mathbb{N}$, so 23 is a tree over $\mathbb{N}$ by the first check.

$\begin{array}{ccc} & 10 & \\ / & & \backslash \\ 14 & & 78 \end{array}$ is of the form $\begin{array}{ccc} & p & \\ / & & \backslash \\ t_1 & & t_2 \end{array}$, where p is 10 , t_1 is 14, and t_2 is 78

Now, obviously $10 \in \mathbb{N}$, so $p \in S$.
Also, $14 \in \mathbb{N}$ and $78 \in \mathbb{N}$, so both pass by the first check.

Contents

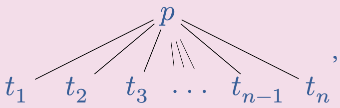
§1.8.4. Structural Induction

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

≑ **structural induction for trees**

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

If we can prove the following two statements -

- For each $s \in S$, φ_s is true
- For each tree T of the form 

,

  if φ_{t_1} , φ_{t_2} , φ_{t_3} , … , $\varphi_{t_{n-1}}$ and φ_{t_n} are all true,

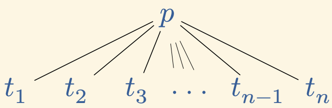
  then φ_T is also true.

then φ_t is true for all trees t over S.

§1.8.5. 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.

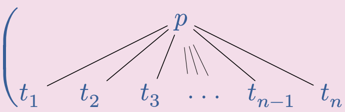
≑ **tree size**

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 times the elements of S appear in a tree over S.

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

size (

) $:= 1 + \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + \ldots + \text{size}(t_{n-1}) + \text{size}(t_n)$

§1.8.6. Termination

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

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

Proof For each tree t, let φ_t be the statement

" The definition of size(t) terminates "

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

- **⟨⟨ $\forall s \in S, \varphi_s$ is true ⟩⟩**
  size(s) := 1, so the definition terminates immediately.
- **⟨⟨ For each tree T of the form … then φ_T is also true⟩⟩**
  Assume that each of $\varphi_{t_1}, \varphi_{t_2}, \varphi_{t_3}, \ldots, \varphi_{t_{n-1}}, \varphi_{t_n}$ is true.
  That means that each of $\text{size}(t_1), \text{size}(t_2), \text{size}(t_3), \ldots, \text{size}(t_{n-1}), \text{size}(t_n)$ will terminate.

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

  Thus, we can see that each term in the right-hand side terminates.
  Therefore, the left-hand side "size(T)",
  being defined as an addition of these terms,
  must also terminate.
  (since addition of finitely many terminating terms always terminates)

  Hence φ_T is proved!

Hence the theorem is proved!! ■

tree depth

Fix a set S.

≑ tree depth

depth : Trees over $S \to \mathbb{N}$

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

$$\text{depth}\left(\begin{array}{cccccc} & & p & & & \\ t_1 & t_2 & t_3 & \cdots & t_{n-1} & t_n \end{array}\right) := 1 + \max_{1 \le i \le n} \{\text{depth}(t_i)\}$$

1. Prove that the definition of the function "depth" terminates on any tree over S.
2. Prove that for any tree t over the set S,
$$\text{depth}(t) \le \text{size}(t)$$
3. When is $\text{depth}(t) == \text{size}(t)$?

Exercise

This exercise is optional as it can be difficult, but it can be quite illuminating to understand the solution. So even if you don't solve it, you should ask for a solution from someone.

Using the ≑ **principle of mathematical induction**,

prove ≑ **structural induction for trees**.

§1.9. Why Trees?

But why care so much about trees anyway? Well, that is mainly due to the previously mentioned fact - "**In fact, any object in Haskell is internally 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.

§1.9.1. The Problem

Suppose we are given that $x = 5$ and then asked to find out the value of the expression $x^3 \cdot x^5 + x^2 + 1$.

How can we do this?

Well, since we know that $x^3 \cdot x^5 + x^2 + 1$ is the function $+$ applied to the inputs $x^3 \cdot x^5$ and $x^2 + 1$, we can first find out the values of these inputs and then apply $+$ on them!

Similarly, as long as we can put an expression in the form $f(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$, we can find out its value by finding out the values of its inputs and then applying f on these values.

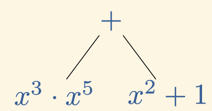
So, for dumb Haskell to do this (figure out the values of expressions, which is quite an important ability) , a vital requirement is to be able to easily put expressions in the form $f(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$.

But this can be quite difficult - In $x^3 \cdot x^5 + x^2 + 1$, it takes our human eyes and reasoning to figure it out fully, and for long, complicated expressions it will be even harder.

§1.9.2. The Solution

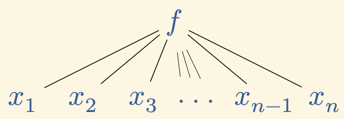
One way to make this easier to represent the expression in the form of a tree -

For example, if we represent $x^3 \cdot x^5 + x^2 + 1$ as

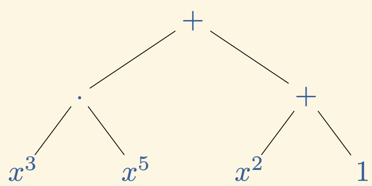


, it becomes obvious what the function is and what the inputs are to which it is applied.

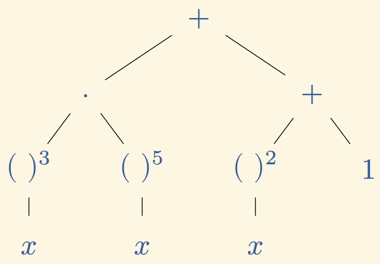
In general, we can represent the expression $f(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$ as



But why stop there, we can represent the sub-expressions (such as $x^3 \cdot x^5$ and $x^2 + 1$) as trees too -



and their sub-expressions can be represented as trees as well -



This is known as the as an Abstract Syntax Tree, and this is (approximately) how Haskell stores expressions, i.e., how it stores everything.

abstract syntax tree

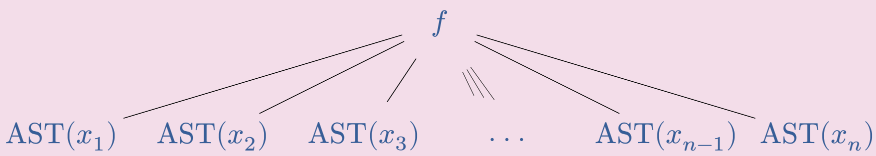
The **abstract syntax tree of a well-formed expression** is defined by applying the "function" **AST** to the expression.

The "function" **AST** is defined as follows -

$\text{AST} : \text{Expressions} \to \text{Trees over values and variables}$

$\text{AST}(v) := v, \text{if } v \text{ is a value or variable}$

$\text{AST}(f(x_1, x_2, x_3, ..., x_{n-1}, x_n)) :=$



§1.9.3. Exercises

All the following exercises are optional, as they are not the most relevant for concept-building. They are just a collection of problems we found interesting and arguably solvable with the theory of this chapter. Have fun![3]

Turbo The Snail(IMO 2024, P5)

Turbo the snail is in the top row of a grid with $s \geq 4$ rows and $s - 1$ columns and wants to get to the bottom row. However, there are $s - 2$ hidden monsters, one in every row except the first and last, with no two monsters in the same column. Turbo makes a series of attempts to go from the first row to the last row. On each attempt, he chooses to start on any cell in the first row, then repeatedly moves to an orthogonal neighbor. (He is allowed to return to a previously visited cell.) If Turbo reaches a cell with a monster, his attempt ends and he is transported back to the first row to start a new attempt. The monsters do not move between attempts, and Turbo remembers whether or not each cell he has visited contains a monster. If he reaches any cell in the last row, his attempt ends and Turbo wins.

Find the smallest integer n such that Turbo has a strategy which guarantees being able to reach the bottom row in at most n attempts, regardless of how the monsters are placed.

[3]Atleast one author is of the opinion:

All questions are clearly compulsory and kids must write them on paper using quill made from flamingo feathers to hope to understand anything this chapter teaches.

X Points in Triangle

Inside a right triangle a finite set of points is given. Prove that these points can be connected by a broken line such that the sum of the squares of the lengths in the broken line is less than or equal to the square of the length of the hypotenuse of the given triangle.

X Joining Points(IOI 2006, 6)

A number of red points and blue points are drawn in a unit square with the following properties:

- The top-left and top-right corners are red points.
- The bottom-left and bottom-right corners are blue points.
- No three points are collinear.

Prove it is possible to draw red segments between red points and blue segments between blue points in such a way that: all the red points are connected to each other, all the blue points are connected to each other, and no two segments cross.

As a bonus, try to think of a recipe or a set of instructions one could follow to do so.

Hint: Try using the 'trick' you discovered in X **Points in Triangle**.

X Usmions(USA TST 2015, simplified)

A physicist encounters 2015 atoms called usamons. Each usamon either has one electron or zero electrons, and the physicist can't tell the difference. The physicist's only tool is a diode. The physicist may connect the diode from any usamon A to any other usamon B. (This connection is directed.) When she does so, if usamon A has an electron and usamon B does not, then the electron jumps from A to B. In any other case, nothing happens. In addition, the physicist cannot tell whether an electron jumps during any given step. The physicist's goal is to arrange the usamons in a line such that all the charged usamons are to the left of the uncharged usamons, regardless of the number of charged usamons. Is there any series of diode usage that makes this possible?

X Battery

(a) There are $2n + 1 (n > 2)$ batteries. We don't know which batteries are good and which are bad but we know that the number of good batteries is greater by 1 than the number of bad batteries. A lamp uses two batteries, and it works only if both of them are good. What is the least number of attempts sufficient to make the lamp work?

(b) The same problem but the total number of batteries is $2n (n > 2)$ and the numbers of good and bad batteries are equal.

X Seven Tries (Russia 2000)

Tanya chose a natural number $X \leq 100$, and Sasha is trying to guess this number. He can select two natural numbers M and N less than 100 and ask about $\gcd(X + M, N)$. Show that Sasha can determine Tanya's number with at most seven questions.

Note: We know of atleast 5 ways to solve this. Some can be generalized to any number k other than 100, with $\lceil \log_2(k) \rceil$ many tries, other are a bit less general. We hope you can find atleast 2.

X The best (trollest) codeforces question ever! (Codeforces 1028B)

Let $s(k)$ be sum of digits in decimal representation of positive integer k. Given two integers $1 \leq m, n \leq 1129$ and n, find two integers $1 \leq a, b \leq 10^{2230}$ such that

- $s(a) \geq n$
- $s(b) \geq n$
- $s(a + b) \leq m$

For Example

Input1 : 6 5

Output1 : 6 7

Input2 : 8 16

Output2 : 35 53

X Rope

Given a $r \times c$ grid with $0 \leq n \leq r * c$ painted cells, we have to arrange ropes to cover the grid. Here are the rules through example:

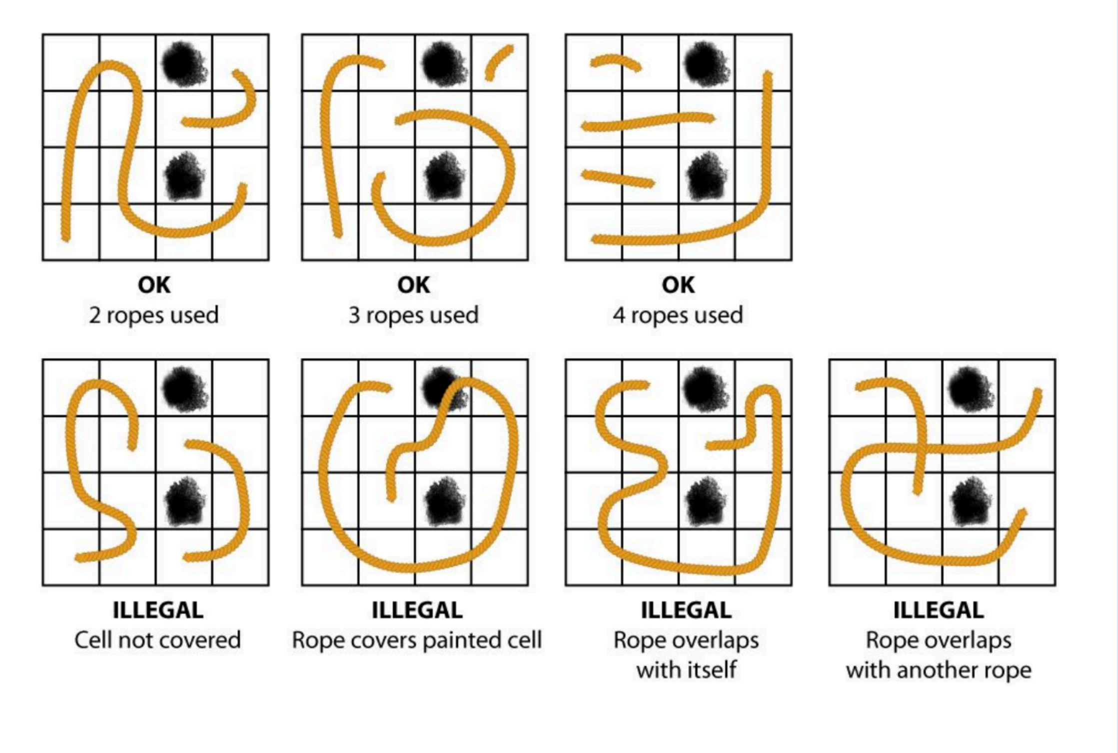


Figure out an algorithm/recipe to covering the grid using $n + 1$ ropes legally.

Hint: Try to first do the $n = 0$ case. Then $r = 1$ case, with arbitrary n. Does this help?

X n composite

Given N, find N consecutive integers that are all composite numbers.

X Divided by 5^n

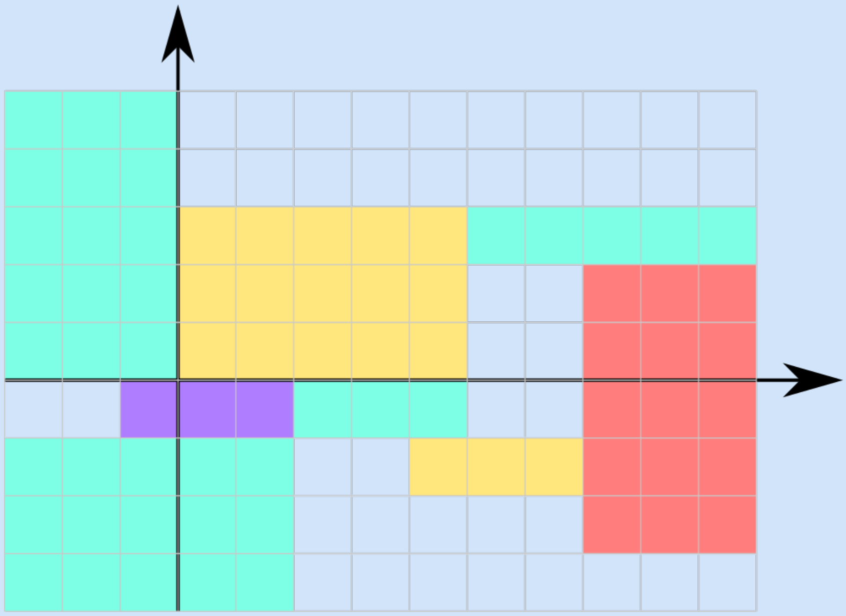
Prove that for every positive integer n, there exists an n-digit number divisible by 5^n, all of whose digits are odd.

X This was rated 2100? (Codeforces 763B)

One of Timofey's birthday presents is a colourbook in the shape of an infinite plane. On the plane, there are n rectangles with sides parallel to the coordinate axes. All sides of the rectangles have odd lengths. The rectangles do not intersect, but they can touch each other.

Your task is, given the coordinates of the rectangles, to help Timofey color the rectangles using four different colors such that any two rectangles that **touch each other by a side** have **different colors**, or determine that it is impossible.

For example,



is a valid filling. Make an algorithm/recipe to fulfill this task.

PS: You will feel a little dumb once you solve it.

X Seating

Wupendra Wulkarni storms into the exam room. He glares at the students.

"Of course you all sat like this on purpose. Don't act innocent. I know you planned to copy off each other. Do you all think I'm stupid? Hah! I've seen smarter chairs.

Well, guess what, darlings? I'm not letting that happen. Not on my watch.

Here's your punishment - uh, I mean, assignment:

You're all sitting in a nice little grid, let's say n rows and m columns. I'll number you from 1 to $n \cdot m$, row by row. That means the poor soul in row i, column j is student number $(i-1) \cdot m + j$. Got it?

Now, you better rearrange yourselves so that none of you little cheaters ends up next to the same neighbor again. Side-by-side, up-down—any adjacent loser you were plotting with in the original grid? Yeah, stay away from them."

Your task is this: Find a new seating chart (in general an algorithm/recipie), using n rows and m columns, using every number from 1 to $n \cdot m$ such that no two students who were neighbors in the original grid are neighbors again.

And if you think it's impossible, then prove it as Wupendra won't satisfy for anything less.

X Yet some more Fibonacci Identity

Fibonacci sequence is defined as $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

(i) Prove that

$$\sum_{n=2}^{\infty} \arctan\left(\frac{(-1)^n}{F_2 n}\right) = \frac{1}{2} \arctan\left(\frac{1}{2}\right)$$

Hint : What is this problem doing on this list of problems?

(ii) Every natural number can be expressed uniquely as a sum of Fibonacci numbers where the Fibonacci numbers used in the sum are all distinct, and no two consecutive Fibonacci numbers appear.

(iii) Evaluate

$$\sum_{i=2}^{\infty} \frac{1}{F_{i-1} F_{i+1}}$$

X Round Robin

A group of n people play a round-robin chess tournament. Each match ends in either a win or a lost. Show that it is possible to label the players $P_1, P_2, P_3, \cdots, P_n$ in such a way that P_1 defeated P_2, P_2 defeated $P_3, \ldots, P_{n-1}$ defeated P_n.

Stamps

(i) The country of Philatelia is founded for the pure benefit of stamp-lovers. Each year the country introduces a new stamp, for a denomination (in cents) that cannot be achieved by any combination of older stamps. Show that at some point the country will be forced to introduce a 1-cent stamp, and the fun will have to end.

(ii) Two officers in Philatelia decide to play a game. They alternate in issuing stamps. The first officer to name 1 or a sum of some previous numbers (possibly with repetition) loses. Determine which player has the winning strategy.

Seven Dwarfs

The Seven Dwarfs are sitting around the breakfast table; Snow White has just poured them some milk. Before they drink, they perform a little ritual. First, Dwarf 1 distributes all the milk in his mug equally among his brothers' mugs (leaving none for himself). Then Dwarf 2 does the same, then Dwarf 3, 4, etc., finishing with Dwarf 7. At the end of the process, the amount of milk in each dwarf's mug is the same as at the beginning! What was the ratio of milt they started with?

Coin Flip Scores

A gambling graduate student tosses a fair coin and scores one point for each head that turns up and two points for each tail. Prove that the probability of the student scoring exactly n points at some time in a sequence of n tosses is $\frac{2+\left(-\frac{1}{2}\right)^n}{3}$

Coins (IMO 2010 P5)

Each of the six boxes B_1, B_2, B_3, B_4, B_5, B_6 initially contains one coin. The following operations are allowed

(1) Choose a non-empty box B_j, $1 \leq j \leq 5$, remove one coin from B_j and add two coins to B_{j+1};

(2) Choose a non-empty box B_k, $1 \leq k \leq 4$, remove one coin from B_k and swap the contents (maybe empty) of the boxes B_{k+1} and B_{k+2}.

Determine if there exists a finite sequence of operations of the allowed types, such that the five boxes B_1, B_2, B_3, B_4, B_5 become empty, while box B_6 contains exactly $2010^{2010^{2010}}$ coins.

Installing Haskell

§2.1. Installation

§2.1.1. General Instructions

1. This may take a while, so make sure that you have enough time on your hands.
2. Make sure that your device has enough charge to last you the entire installation process.
3. Make sure that you have a strong and stable internet connection.
4. Make sure that any antivirus(es) that you have on your device is fully turned off during the installation process. You can turn it back on immediately afterwards.
5. Make sure to follow the following instructions **IN ORDER**.
   Make sure to **COMPLETE EACH STEP** fully **BEFORE** moving on to the **NEXT STEP**.

§2.1.2. Choose your Operating System

§2.1.2.1. Linux

1. **Install Haskell**
   1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
   2. Close all open windows and running processes other than wherever you are reading this.
   3. Open the directory `Haskell/installation/Linux` in your text editor.
      (We have more support for Visual Studio Code, but any text editor should do)
   4. Type in the commands in the `installHaskell` file into the terminal.
   5. This may take a while.
   6. You will know installation is complete at the point when it says `Press any key to exit`.
   7. Restart (shut down and open again) your device.
2. **Install HaskellSupport**
   1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
   2. Close all open windows and running processes other than wherever you are reading this.
   3. Open the directory `Haskell/installation/Linux` in your text editor.
      (We have more support for Visual Studio Code, but any text editor should do)

4. Type in the commands in the `installHaskellSupport` file in the terminal.
5. This may take a while.
6. You will know installation is complete at the point when it says `Press any key to Exit`.
7. Restart (shut down and open again) your device.

§2.1.2.2. MacOS

1. **Install Haskell**
   1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
   2. Close all open windows and running processes other than wherever you are reading this.
   3. Open the folder `Haskell` in Finder .
   4. Open the folder `installation` in Finder.
   5. Right click on the folder `MacOS` in Finder, and select `Open in Terminal`.
   6. Type in `chmod +x installHaskell.command` in the terminal.
   7. Close the terminal window.
   8. Open the folder `MacOS` in Finder.
   9. Double-click on `installHaskell.command`.
   10. This may take a while.
   11. You will know installation is complete at the point when it says `Press any key to exit`.
   12. Restart (shut down and open again) your device.
2. **Install Visual Studio Code**
   Get it *here*.
3. **Install HaskellSupport**.
   1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
   2. Close all open windows and running processes other than wherever you are reading this.
   3. Open the folder `Haskell` in Finder .
   4. Open the folder `installation` in Finder.
   5. Right click on the folder `MacOS` in Finder, and select `Open in Terminal`.
   6. Type in `chmod +x installHaskellSupport.command` in the terminal.
   7. Close the terminal window.
   8. Open the folder `MacOS` in Finder.
   9. Double-click on `installHaskellSupport.command`.
   10. This may take a while.
   11. You will know installation is complete if a new window pops up asking whether you trust authors. Click on "Trust".

12. Restart (shut down and open again) your device.

§2.1.2.3. Windows

1. **Install Haskell**.
    1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
    2. Close all open windows and running processes other than wherever you are reading this.
    3. Open the folder `Haskell` in File Explorer .
    4. Open the folder `installation` in File Explorer.
    5. Open the folder `Windows` in File Explorer.
    6. Double-click on `installHaskell`.
    7. This may take a while.
    8. You will know installation is complete at the point when it says `Press any key to exit`.
    9. Restart (shut down and open again) your device.
2. **Install Visual Studio Code**
   Get it *here*.
3. **Install HaskellSupport**.
    1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
    2. Close all open windows and running processes other than wherever you are reading this.
    3. Open the folder `Haskell` in File Explorer.
    4. Open the folder `installation` in File Explorer.
    5. Open the folder `Windows` in File Explorer.
    6. Double-click on `installHaskellSupport`.
    7. This may take a while.
    8. You will know installation is complete if a new window pops up asking whether you trust authors. Click on “Trust”.
    9. Restart (shut down and open again) your device.

§2.2. Running Haskell

Open VS Code. A window “Welcome” should be open right now. If you close that tab, then a tab with `helloWorld` written should pop up.

If you right-click on `True`, a drop-down menu should appear, in which you should select “Run Code”.

You have launched GHCi. After some time, you should see the symbol `>>>` appear.

Type in `helloWorld` after the `>>>`.

It should reply `True`.

§2.3. Fixing Errors

If you see squiggly red, yellow, or blue lines under your text, that means there is an error, warning, or suggestion respectively.

To explore your options to remedy the issue, put your text cursor at the text and click `Ctrl+.` .

You have opened the QuickFix menu.

You can now choose a suitable option.

§2.4. Autocomplete

Just like texting with your friends, VS Code also gives you useful auto-complete options while you are writing.

To navigate the auto-complete options menu, hold down the `Ctrl` key while navigating using the ↑ and ↓ keys.

To accept a particular auto-complete suggestion, use `Ctrl+Enter`.

Basic Syntax

We will now gradually move to actually writing in Haskell. Programmers refer to this step as learning the "syntax" of a language.

To do this we will slowly translate the syntax of mathematics into the corresponding syntax of Haskell.

§3.1. The Building Blocks

Just like in math, the Haskell language relies on the symbols and expressions. The symbols include whatever characters can be typed by a keyboard, like `q`, `w`, `e`, `r`, `t`, `y`, `%`, `(`, `)`, `=`, `1`, `2`,etc.

§3.2. Values

Haskell has values just like in math.

≑ **value**

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

Examples include -

- The number `pi` with the decimal expansion `3.141592653589793 ...`
- The order `<` on the `Integer`s
- The function of squaring an `Integer`
- the character `'a'` from the keyboard
- `True` and `False`

§3.3. Variables

Haskell also has its own variables.

≑ **variable**

A **variable** is a symbol or chain of symbols,
meant to represent an arbitrary object of some ***type***,
usually as a way to show that whatever process follows is general enough so that the process can be carried out with ***any arbitrary value*** from that ***type***.

The following examples should clarify further.

We have previously seen how variables are used in function definitions and theorems.

Even though we can prove theorems about Haskell, the Haskell language itself supports only function definitions and not theorems.

So we can use variables in function definitions. For example -

λ **double**

```
double :: Integer → Integer
double x = x + x
```

This reads - "`double` is a function that takes an `Integer` as input and gives an `Integer` as output. The `double` of an input `x` is the output `x + x`"

Note that `x` here is a variable.

Also, in mathematics we would write `double`(`x`), but Haskell does not need those brackets.
So we can simply put some space between `double` and `x`, i.e.,
we write `double x`,
in order to indicate that `double` is the name of the function and `x` is its input.

Also, **Note** that the names of Haskell ≑ **variable**s have to begin with an lowercase English letter.

§3.4. Types

Every ≑ **value** and ≑ **variable** in Haskell must have a "type".

For example,

- `'a'` has the type `Char`,
  indicating that it is a character from the keyboard.
- `5` can have the type `Integer`,
  indicating that it is an integer.
- `double` has the type `Integer → Integer`,
  indicating that is a function that takes an integer as input and gives an integer as output.
- In the definition of `double`, specifically "`double x = x + x`",
  the variable `x` has type `Integer`,
  indicating that it is an integer.

The type of an object is like a short description of the object's "nature".

Also, **Note** that the names of types usually have to begin with an uppercase English letter.

§3.4.1. Using GHCi to get Types

GHCi allows us to get the type of any value using the command `:type +d` followed by the value -

λ :type +d

```
>>> :type +d 'a'
'a' :: Char

>>> :type +d 5
5 :: Integer

>>> :type +d double
double :: Integer → Integer
```

`x :: T` is just Haskell's way of saying "`x` is of type `T`".

Note - The `+d` at the end of `:type +d` stands for "default", which means that its a more basic version of the more powerful command `:type`

For example -

```
>>> :type +d (+)
(+) :: Integer → Integer → Integer
```

This reads - “The function `+` takes in two `Integer`s as inputs and gives an `Integer` as output”

Or more generally -

λ :type

```
>>> :type (+)
(+) :: Num a ⇒ a → a → a
```

This reads - “The function `+` takes in two `Num`bers as inputs and gives a `Num`ber as output”

In summary, `:type +d` is specific, whereas `:type` is general.

For now, we will be assuming `:type +d` throughout, until we get to Chapter 6.

§3.4.2. Types of Functions

As we have seen before, `double` has type `Integer → Integer`. This function has a single input.

And the “basic” type of the ≑ **infix binary operator** `+` is `Integer → Integer → Integer`. This function has two inputs.

We can also define functions which takes a greater number of inputs -

λ functions with many inputs

```
sumOf2 :: Integer → Integer → Integer
sumOf2 x y = x + y
-- The above function has 2 inputs

sumOf3 :: Integer → Integer → Integer → Integer
sumOf3 x y z = x + y + z
-- The above function has 3 inputs

sumOf4 :: Integer → Integer → Integer → Integer → Integer
sumOf4 x y z w = x + y + z + w
-- The above function has 4 inputs
```

So we can deduce that in general,

if a function takes n inputs of types `T1`, `T2`, `T3`,..., `Tn` respectively,

and gives an output of type `T`,

then the function itself will have type `T1 → T2 → T3 → . . . → Tn → T`.

§3.5. Well-Formed Expressions

Of course, since we have ≑ **value**s and ≑ **variable**s, we can define “well-formed expressions” in a very similar manner to what we had before -

Contents

≑ **checking whether expression is well-formed**

It is difficult to give a direct definition of a **well-formed expression**.

So before giving the direct definition,
we define a ***formal procedure*** to check whether an expression is a **well-formed expression** or not.

The procedure is as follows -

Given an expression e,

- first check whether e is
  - a ≑ **value**, or
  - a ≑ **variable**

  in which cases e passes the check and is a **well-formed expression**.

Failing that,

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

And only if e passes this check will it be a **well-formed expression**.

≑ **well-formed expression**

An ***expression*** is said to be a **well-formed expression** if and only if it passes the formal checking procedure defined in ≑ **checking whether expression is well-formed**.

Recall, that last time in Section §1.5., when we were formally checking that $x^3 \cdot x^5 + x^2 + 1$ is indeed a ≑ **well-formed expression**, we skipped the part about checking whether

"$e_1, e_2, e_3, ..., e_n$ are ... *valid inputs* to f."

which is present in the very last part of the formal procedure
≑ **checking whether mathematical expression is well-formed**.

That is, we didn't have a very good way to check whether

$$\text{the input to a function} \in \text{the domain of the function}$$

, Thus we could potentially face mess-ups like

$$(1, 2) + 3$$

Here, the expression is not well-formed because $(1, 2)$ is not a valid input for $+$
(in other words $(1, 2) \notin$ the domain of $+$)
, but we had no way to prevent this before.

Now, with types, this problem is solved!

If a function has type `T1 -> T2`,
and Haskell wants to check whether whatever input has been given to it is a valid input or not,
it need only check that this input is of type `T1`.

We can see this in action with `double` -

```
>>> double 12
24
```

`12` has type `Integer`, and therefore Haskell is quite happy to take it as input to the function `double` of type `Integer -> Integer`.

However -

```
>>> double 'a'

<interactive>:1:8: error: [GHC-83865]
    * Couldn't match expected type `Integer' with actual type `Char'
    * In the first argument of `double', namely 'a'
      In the expression: double 'a'
      In an equation for `it': it = double 'a'
```

Since `double` has type `Integer -> Integer`, Haskell tries to check whether the input `'a'` has type `Integer`, but discovers that it actually has a different type (`Char`), and therefore disallows it.

This is actually the point of types, and the consequences are very powerful.
Why? Recall that ≑ **well-formed expressions** are supposed to be only those expressions which are meaningful. Since Haskell has the power to check whether expressions are well-formed or not, it will never allow us to write a "meaningless" expression.

Other programming languages which don't have types allows one to write these "meaningless" expressions and that creates "bugs" a.k.a logical errors.

The very powerful consequence is that Haskell manages to **provably avoid any of these logical errors**!

§3.6. Infix Binary Operators

If we enclose **an** ≑ **infix binary operator** in **brackets**, we can use it just as we would a **function**

λ using infix operator as function

```
>>> 12 + 34 -- usage as infix binary operator
46
>>> (+) 12 34 -- usage as a normal Haskell function
46

>>> 12 - 34 -- as infix binary operator
-22
>>> (-) 12 34 -- usage as a normal Haskell function
-22

>>> 12 * 34 -- as infix binary operator
408
>>> (*) 12 34 -- usage as a normal Haskell function
408
```

Conversely, if we enclose a **function** in **backticks** (`` ` ``), we can use it just like an ≑ **infix binary operator**.

λ using function as infix operator

```
>>> f x y = x*y + x + y -- function definition
>>> f 3 4 -- usage as a normal Haskell function
19
>>> 3 `f` 4 -- usage as an infix binary operator
19
```

§3.6.1. Precedence

÷ **infix binary operators** sometimes introduce a small complication.

For example, when we write `a + b * c`,

do we mean `a + ( b * c )`

or do we mean `( a + b ) * c` ?

We know that the method to solve these problems are the BODMAS or PEMDAS conventions.

So Haskell assumes the first option due to BODMAS or PEMDAS conventions, whichever one takes your fancy.

This problem is called the problem of "precedence", i.e.,
"which operations in an expression are meant to be applied first (preceding) and which to be applied later?"

Haskell has a convention for handling all possible ÷ **infix binary operators** that extends the PEMDAS convention.
(It assigns to each ÷ **infix binary operator** a number indicating the precedence, and those with greater value of precedence are evaluated first)

But there still remains an issue -

What about `a - b - c` ?

Does it mean `( a - b ) - c` ,

or does it mean `a - ( b - c )` ?

Observe that this issue is not solved by the BODMAS or PEMDAS convention.

Haskell chooses `( a - b ) - c`, because `-` is "left-associative".

÷ **left-associative**

If an ÷ **infix binary operator** ? is **left-associative**, it means that the expression

```
x₁ ? x₂ ? x₃ ? ... ? xₙ
```

is equivalent to

```
( x₁ ? x₂ ) ? x₃ ? ... ? xₙ
```

which means that the leftmost ? is evaluated first.

Therefore `a - b - c` is equivalent to `( a - b ) - c` and not `a - ( b - c )`.

But what about `a - b - c - d` ?

```
a - b - c - d
-- take ? as -, n as 4, x₁ as a, x₂ as b, x₃ as c, x₄ as d
== ( a - b ) - c - d
-- take ? as -, n as 3, x₁ as ( a - b ), x₂ as c, x₃ as d
== ( ( a - b ) - c ) - d
```

x order of operations

Find out the value of `7 - 8 - 4 - 15 - 65 - 42 - 34`

We also have the complementary notion of being "right-associative".

÷ right-associative

If an ÷ **infix binary operator** `?` is **right-associative**, it means that the expression

```
x₁ ? x₂ ? x₃ ? ... ? xₙ₋₂ ? xₙ₋₁ ? xₙ
```

is equivalent to

```
x₁ ? x₂ ? x₃ ? ... ? xₙ₋₂ ? ( xₙ₋₁ ? xₙ )
```

which means that the rightmost `?` is evaluated first.

§3.7. Logic

§3.7.1. Truth

The way to represent truth or falsity in Haskell is to use the value `True` or the value `False` respectively. Both values are of type `Bool`.

```
>>> :type True
True :: Bool

>>> :type False
False :: Bool
```

The `Bool` type means "true or false".

The values `True` and `False` are called `Bool`eans.

§3.7.2. Statements

Haskell can check the correctness of some very simple mathematical statements -

λ simplest logical statements
```
>>> 1 < 2
True

>>> 2 < 1
False

>>> 5 == 5
True

>>> 5 /= 5
False

>>> 4 == 5
False

>>> 4 /= 5
True
```

Note that `≠` is written as `/=`

Note that `≤` is written as `<=`

etc.

But the very nice fact is that Haskell does not require any new syntax or mechanism for these.

The way Haskell achieves this is an inbuilt ≑ **infix binary operator** named `<`,
which takes two inputs, `x` and `y`,
and outputs `True` if `x` is less than `y`, and otherwise outputs `False`.

So, in the statement `1 < 2`,
the `<` function is given the two inputs `1` and `2`,
and then GHCi evaluates this and outputs the correct value, `True`.

```
>>> 1 < 2
True
```

So let's see if all this makes sense with respect to the type of `<` -

λ type of <
```
>>> :type (<)
(<) :: Ord a => a -> a -> Bool
```

Indeed we see that `<` takes two inputs of type `a`, and gives an output of type `Bool`.

§3.8. Conditions

So we can use these functions to define some "condition" on a ≑ **variable**.

For example -

λ condition on a variable
```
isLessThan5 :: Integer -> Bool
isLessThan5 x = x < 5
```

This function encodes the "condition" that the input variable must be less than 5.

However, we would definitely like to express some more complicated conditions as well. For example, we might want to express the condition -

$$x \in (4, 10]$$

We know that $x \in (4, 10]$ if and only both $x > 4$ AND $x \leq 10$ hold true.

Using this fact, we can express the condition "$x \in (4, 10]$" as

```
( x > 4 ) && ( x ≤ 10 )
```

in Haskell, since `&&` represents "AND" in Haskell.

Let's take `x` to be `7` and see what is happening here step by step -

```
  ( x > 4 ) && ( x ≤ 10 )
= ( 7 > 4 ) && ( 7 ≤ 10 )
=    True   && ( 7 ≤ 10 )
=    True   &&    True
-- now applying the definition of && aka AND
= True
```

which is correct since " $7 \in (4, 10]$ " is indeed a true statement.

So the type of `&&` is -

```
>>> :type (&&)
(&&) :: Bool → Bool → Bool
```

It takes two `Bool`eans as inputs and outputs another `Bool`ean.

§3.8.1. Logical Operators

÷ logical operator

Functions like `&&`, which take in some `Bool`ean(s) as input(s), and give a single `Bool`ean as output are called **logical operators**.

You might have seen some logical operators before with names such AND, OR, NOT, NAND, NOR etc.

As we just saw, they are very useful in combining two conditions into one, more complicated condition.

For example -
- if we want to express the condition

  $$x \in (-\infty, 6] \cup (15, \infty)$$

  , we would re-express it as

  "$x \leq 6$ OR $x > 15$"

  , which could finally be expressed in Haskell as

  ```
  ( x ≤ 6 ) || ( x > 15 )
  ```

  , since `||` is Haskell's way of writing OR.

- if we want to express the condition

  $$x \notin (-\infty, 4)$$

  , we could re-express it as

  $$\text{NOT } (x \in (-\infty, 4))$$

  , which could be further re-written as

  $$\text{NOT } (x < 4)$$

  , which then can be expressed in Haskell as

  `not ( x < 4 )`

We include the definition of `not` as it is quite simple -

λ **not**

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

§3.8.1.1. Exclusive OR aka XOR

Finally, we define a logical operator called XOR.

≑ **XOR**

(boolean$_1$ XOR boolean$_2$) is defined to be true
if and only if
at least one of the 2 inputs is true, but not both,
and otherwise is defined to be false.

Suppose P and Q are two people running a race against each other.

Then at least one of them will win, but not both.

Therefore ((A wins) XOR (B wins)) would be true.

Also, (false XOR false) would be false, since at least one of the inputs need to be true.

Finally, (true XOR true) would be false, as both inputs are true.

§3.9. Function Definitions

Functions are a very important tool in mathematics and they form the foundations of Haskell programming.

Nearly everything in Haskell is done using functions, so there various ways of defining many kinds of functions.

§3.9.1. Using Expressions

In its simplest form, a function definition is made up of a `left-hand side` describing the function name and input(s), `=` in the middle and a `right-hand side` describing the output.

An example -

If we want write the following definition

$$f(x, y) := x^3 \cdot x^5 + y^3 \cdot x^2 + 14$$

Then we can write in Haskell -

λ basic function definition

```
f x y = x^3 * x^5 + y^3 * x^2 + 14
```

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 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.

Also, we know that $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

We can include this information in the definition -

λ function definition with explicit type

```
f :: Integer → Integer → Integer
f x y = x^3 * x^5 + y^3 * x^2 + 14
```

Even though it is not mandatory, it is **always** advised to follow the above style and **explicitly provide a particular type** for the function being defined.

Even if an explicit type is not provided, Haskell will assume the most general type the function could have, like what we observed in the `:type` command of GHCi.

Let's try to define ÷ **XOR** in Haskell -

λ xor

```
xor :: Bool → Bool → Bool
xor b1 b2 =
--     at least one of the inputs is True, but   not both
-- ⟺ b1 is True OR b2 is True          , but   not both
-- ⟺ ( b1 == True ) OR ( b2 == True )  , but   not both
-- ⟺ ( b1 == True ) OR ( b2 == True )  , but   not ( b1 AND b2 )
-- ⟺ ( b1 == True ) OR ( b2 == True )  , but ( not ( b1 AND b2 ) )
-- ⟺ ( b1 == True ) OR ( b2 == True )   AND  ( not ( b1 AND b2 ) )
     ( ( b1 == True ) || ( b2 == True ) ) &&  ( not ( b1 && b2 ) )
```

§3.9.2. Some Conveniences

§3.9.2.1. Piecewise Functions

If we have a function definition like

$$
< \text{functionName} > (x) := \begin{cases} < \text{expression}_1 > \ ; \text{if} < \text{condition}_1 > \\ < \text{expression}_2 > \ ; \text{if} < \text{condition}_2 > \\ < \text{expression}_3 > \ ; \text{if} < \text{condition}_3 > \\ . \\ . \\ . \\ < \text{expression}_N > ; \text{if} < \text{condition}_N > \end{cases}
$$

, it can be written in Haskell as

guards

```
functionName
    | condition1 = expression1
    | condition2 = expression2
    | condition3 = expression3
    .
    .
    .
    | conditionN = expressionN
```

For example,

$$
\text{signum} : \mathbb{R} \to \mathbb{R}
$$

$$
\text{signum}(x) := \begin{cases} +1 \ ; \text{if } x \ > \ 0 \\ \ \ 0 \ ; \text{if } x == 0 \\ -1 \ ; \text{if } x \ < \ 0 \end{cases}
$$

can written in Haskell as

basic usage of guards

```
signum :: Double → Double
signum x
    | x >  0 =  1
    | x == 0 =  0
    | x <  0 = -1
```

If a piecewise definition has a “catch-all” or “otherwise” clause at the end, as in

$$
< \text{functionName} > (x) := \begin{cases} < \text{expression}_1 > & ; \text{if} < \text{condition}_1 > \\ < \text{expression}_2 > & ; \text{if} < \text{condition}_2 > \\ < \text{expression}_3 > & ; \text{if} < \text{condition}_3 > \\ . \\ . \\ . \\ < \text{expression}_N > & ; \text{if} < \text{condition}_N > \\ < \text{expression}_{N+1} > & ; \text{otherwise} \end{cases}
$$

, it can be written in Haskell as

guards

```
functionName
    | condition1 = expression1
    | condition2 = expression2
    | condition3 = expression3
    .
    .
    .
    | conditionN = expressionN
    | otherwise  = expression(N+1)
```

This `|` syntax symbol is called a "guard".

For example -

otherwise

```
xor1 :: Bool → Bool → Bool
xor1 b1 b2
    | (not b1) && (not b2) = False -- when none of the inputs are True
    | b1 && b2             = False -- when both of the inputs are True
    | otherwise            = True  -- any other situation
```

If a piecewise definition has ony two parts

$$
< \text{functionName} > (x) := \begin{cases} < \text{expression}_1 > ; \text{if} < \text{condition} > \\ < \text{expression}_2 > ; \text{otherwise} \end{cases}
$$

then a lot programming languages have a simple construct called "if-else" to express this -

if-then-else

```
functionName = if condition then expression1 else expression2
```

For example -

if-then-else example

```
xor2 :: Bool → Bool → Bool
xor2 b1 b2 = if b1 == b2 then False else True
-- if both inputs to xor are the same, then output False, otherwise True
```

§3.9.2.2. Pattern Matching

We can write the map of every possible input one by one. This is called "exhaustive pattern matching".

λ exhaustive pattern matching

```
xor3 :: Bool → Bool → Bool -- answer True iff at least one input is True,
but not both
xor3 False False = False -- at least one input should be True
xor3 True  True  = False -- since both inputs are True
xor3 False True  = True
xor3 True  False = True
```

We could be smarter and save some keystrokes.

λ pattern matching

```
xor4 :: Bool → Bool → Bool
xor4 False b = b
xor4 b False = b
xor4 b1 b2 = False
```

Another small pattern match equivalent to `xor1` -

λ unused variables in pattern match

```
xor5 :: Bool → Bool → Bool
xor5 False False = False
xor5 True True = False
xor5 b1 b2 = True
```

But since the variables `b1` and `b2` are not used in the right-hand side,

we can replace them with `_` (read as "wildcard")

λ wildcard

```
xor6 :: Bool → Bool → Bool
xor6 False True = True
xor6 True False = True
xor6 _ _ = False
```

Wildcard (`_`) just means that any pattern will be accepted.

We can use other functions to help us as well -

λ using other functions in RHS

```
xor7 :: Bool → Bool → Bool
xor7 False b = b
xor7 True b = not b
```

We can also piecewise definitions in a pattern match -

λ pattern matches mixed with guards

```
xor8 :: Bool → Bool → Bool
xor8 False b2 = b2 -- Notice, we can have part of the definition unguarded
before entering the guards.
xor8 True b2
    | b2 == False = True
    | b2 == True  = False
```

Now we introduce the `case .. of ..` syntax. It is used to pattern-matching for any expression, not necessarily just the input variables, which are the only kinds of examples we've seen till now.

Contents


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

The case syntax evaluates the `<expression>`, and matches it against each pattern in order. The first matching pattern's corresponding result is returned.

λ trivial case

```
xor9 :: Bool → Bool → Bool
xor9 b1 b2 = case b1 of
  False → b2
  True  → not b2
```

λ non-trivial case

```
xor10 :: Bool → Bool → Bool
xor10 b1 b2 = case ( b1 , b2 ) of
    ( False , False ) → False
    ( True  , True  ) → False
    _                 → True
```

§3.9.2.3. Where, Let

λ where

```
xor11 :: Bool → Bool → Bool
xor11 b1 b2 = atLeastOne && (not both)
    where
        atLeastOne = b1 || b2
        both = b1 && b2
```

λ let

```
xor12 :: Bool → Bool → Bool
xor12 b1 b2 =
    let
        atLeastOne = b1 || b2
        both = b1 && b2
    in
        atLeastOne && (not both)
```

§3.9.2.4. Without Inputs

Let us recall for a moment the definition for `xor2` (in λ **if-then-else example**)

λ if-then-else example

```
xor2 :: Bool → Bool → Bool
xor2 b1 b2 = if b1 == b2 then False else True
-- if both inputs to xor are the same, then output False, otherwise True
```

We can see that this just equivalent to

```
xor13 :: Bool → Bool → Bool
xor13 b1 b2 = not ( b1 == b2 )
```

which can be shortened even further

```haskell
xor14 :: Bool → Bool → Bool
xor14 b1 b2 = b1 /= b2
```

, rewritten by λ **using infix operator as function**

```haskell
xor15 :: Bool → Bool → Bool
xor15 b1 b2 = (/=) b1 b2
```

and thus can finally be shortened to the extreme

λ **function definition without input variables**

```haskell
xor16 :: Bool → Bool → Bool
xor16 = (/=)
```

Notice the curious thing that the above function definition doesn't have any input variables. This ties into a fundamentally important concept called currying which we will explore later.

§3.9.2.5. Anonymous Functions

An anonymous function like

$$(x \mapsto x^3 \cdot x^5 + x^2 + 1) : \mathbb{R} \to \mathbb{R}$$

can written as

λ **basic anonymous function**

```haskell
( \ x → x^3 * x^5 + x^2 + 1 ) :: Double → Double
```

Note that we used $\to$ in place of $\mapsto$,
and also added a `\` (pronounced "lambda") before the input variable.

For an example with multiple inputs, consider

$$\left(x, y \mapsto \frac{1}{x} + \frac{1}{y}\right)$$

which can be written as

λ **multi-input anonymous function**

```haskell
( \ x y → 1/x + 1/y )
```

X **only nand**

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` and `xor` using only `nand`.

§3.9.3. Recursion

A lot of mathematical functions are defined recursively. We have already seen a lot of them in chapter 1 and exercises. Factorial, binomials and fibonacci are common examples.

We can use the recurrence

$$n! := n \cdot (n - 1)!$$

to define the factorial function.

λ factorial

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

We can use the standard Pascal's recurrence

$$\binom{n}{r} := \binom{n-1}{r} + \binom{n-1}{r-1}$$

to define the binomial or "choose" function.

λ binomial

```
choose :: Integer → Integer → Integer
0 `choose` 0 = 1
0 `choose` _ = 0
n `choose` r = (n-1) `choose` r + (n-1) `choose` (r-1)
```

And we have already seen the recurrence relation for the fibonacci function in Section §1.6.3..

λ naive fibonacci definition

```
fib :: Integer → Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

§3.10. Optimization

For fibonacci, note that in λ **naive fibonacci definition** is, well, naive.

This is because we keep recomputing the same values again and again. For example computing `fib 5` according to this scheme would look like:

λ computation of naive fibonacci

```
fib 5 == fib 4 + fib 3
      == (fib 3 + fib 2) + (fib 2 + fib 1)
      == ((fib 2 + fib 1) + (fib 1 + fib 0)) + ((fib 1 + fib 0) + 1)
      == (((fib 1 + fib 0) + 1) + (1 + 1)) + ((1 + 1) + 1)
      == (((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)
      == 8
```

If we can manage to avoid recomputing the same values over and over again, then the computation will take less time.

That is what the following definition achieves.

λ fibonacci by tail recursion

```
fibonacci :: Integer → Integer
fibonacci n = go n 1 1 where
  go 0 a _ = a
  go n a b = go (n - 1) b (a + b)
```

We can see that this is much more efficient. Tracing the computation of `fibonacci 5` now looks like:

λ computation of tail recursion fibonacci

```
fibonacci 5
== go 5 1 1
== go 4 1 2
== go 3 2 3
== go 2 3 5
== go 1 5 8
== go 0 8 13
== 8
```

This is called tail recursion as we carry the tail of the recursion to speed things up. It can be used to speed up naive recursion, although not always.

Another way to evaluate fibonacci will be seen in end of chapter exercises, where we will translate Binet's formula straight into Haskell. Why can't we do so directly? As we can't represent $\sqrt{5}$ exactly and the small errors in the approximation will accumulate due to the number of operations. This exercise should allow you to end up with a blazingly fast algorithm which can compute the 12.5-th million fibonacci number in 1 sec. Our tail recursive formula takes more than 2 mins to reach there.

§3.11. Numerical Functions

÷ Integer and Int

`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.

The reason for `Int` existing is historical. It was the only option at one point and continues to be available for backwards compatibility.

We will assume `Integer` wherever possible.

÷ Rational

`Rational` 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.

`Rational`s are declared using `%` as the vinculum(the dash between numerator and denominator). For example `1%3, 2%5, 97%31`, which respectively correspond to $\frac{1}{3}, \frac{2}{5}, \frac{97}{31}$.

÷ Double

`Double` or Double Precision Floating Point are high-precision approximations of real numbers. For example, consider the "square root" function -

```
>>> sqrt 2 :: Double
1.4142135623730951

>>> sqrt 99999 :: Double
316.226184874055

>>> sqrt 999999999 :: Double
31622.776585872405
```

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 chapter 10. For now, assume that these functions work exactly as expected.

`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
```

§3.11.1. Division, A Trilogy

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

`recip` is a function which reciprocates a given number, but it has rather interesting type signature. It is only defined on types with the `Fractional` "type-class". 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`.[4]

However, suppose that we want to do integer division and we want a quotient and remainder.

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

[4]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 automatically infer the reciprocation. So technically, for a datatype to be a member of the type class `Fractional` it needs to have either reciprocation or division defined, the other is inferred.

```
>>> 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.

X Div vs Quot

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
```

Mod vs Rem

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 similarly when the divisor and dividend are of the same sign, they seem to diverge when the signs don't match.

The thing here is we always want our division algorithm to satisfy $d * q + r = n, |r| < |d|$ where d is the divisor, 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 division value (recall $\lfloor -3.56 \rfloor = -4$) while `quot` returns the truncated value of the true division (recall $\text{truncate}(-3.56) = -3$ as we are just truncating the decimal point off). The reason we keep both of them in Haskell is to be comfortable 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 exercise for us now would be implementing our very own integer division algorithm. We begin with a division algorithm for only positive integers.

Contents

λ A division algorithm on positive integers by repeated 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 negatives 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 = divide 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)
```

X **Another Division**

Figure out which kind of division have we implemented above, floored or truncated.

Now implement the other one yourself by modifying the above code appropriately.

§3.11.2. Exponentiation

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

Can you see the difference?

What can we say about the three exponentiation operators?

```
>>> a = 5 :: Int
>>> b = 0.5 :: Float
>>>
>>> a^a
3125
>>> a^^a
<interactive>:4:2: error: [GHC-39999]
>>> a**a
<interactive>:5:2: error: [GHC-39999]
>>>
>>> a^b
<interactive>:6:2: error: [GHC-39999]
>>> a^^b
<interactive>:7:2: error: [GHC-39999]
>>> a**b
<interactive>:8:4: error: [GHC-83865]
>>>
>>> b^a
3.125e-2
>>> b^^a
3.125e-2
>>> b**a
<interactive>:11:4: error: [GHC-83865]
>>>
>>> b^b
<interactive>:12:2: error: [GHC-39999]
>>> b^^b
<interactive>:13:2: error: [GHC-39999]
>>> b**b
0.70710677
>>>
>>> a^(-a)
*** Exception: Negative exponent
>>> a^^(-a)
<interactive>:16:2: error: [GHC-39999]
>>> a**(-a)
<interactive>:17:2: error: [GHC-39999]
>>>
>>> b^(-a)
*** Exception: Negative exponent
>>> b^^(-a)
32.0
>>> b**(-a)
<interactive>:20:6: error: [GHC-83865]
```

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

- `^` is raising general numbers to positive integral powers. This means it makes no assumptions about if the base can be reciprocated and just produces an exception if the power is negative and error if the power is fractional.

- `^^` 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 exponentiation.

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 accuracy and may even output `Infinity` in some cases. The `...` means I am truncating the output for readability, GHCi did give the complete 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 exponentiation algorithm**

```
exponentiation :: (Num a, Integral b) => a -> b -> a
exponentiation a 0 = 1
exponentiation a b = if b < 0
  then error "no negative exponentiation"
  else a * (exponentiation a (b-1))
```

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

However, we could do a bit better here. Notice, to evaluate a^b, we are making b multiplications.

A fact, which we shall prove in chapter 10, is that multiplication of big numbers is faster when it is balanced, that is the numbers being multiplied have similar 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 recursively till we have the solution.

λ **A better exponentiation algorithm using divide and conquer**

```
exponentiation :: (Num a, Integral b) => a -> b -> a
exponentiation a 0 = 1
exponentiation a b
  | b < 0     = error "no negative exponentiation"
  | even b    = let half = exponentiation a (b `div` 2)
                in half * half
  | otherwise = let half = exponentiation 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 at most 3 multiplications at each step and there are at most $\lceil \log_2(b) \rceil$ steps. This brings us down from b multiplications to $3 \log(b)$ multiplications. Furthermore, most of these multiplications are somewhat balanced and hence optimized.

This kind of a strategy 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, in searching through data, in even solving differential equations and training AI models) and we will see it again shortly.

§3.11.3. `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.

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 maximal-ity 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**

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


lcm :: Integer → Integer → Integer
lcm p q = (p * q) `div` (gcd p q)
```

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

This algorithm predates computers by approximately 2300 years. It was first described 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 develop.

§3.11.4. 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 important 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 :: Char -> Int` takes a character and returns its corresponding numeric code. The function `chr :: Int -> Char` does the inverse: it takes a number and returns the character it represents.

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

§3.12. Mathematical Functions

We will now talk about mathematical functions like `log`, `sqrt`, `sin`, `asin` etc. We will also take this opportunity to talk about real exponentiation. 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 functions 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?

§3.12.1. Binary Search

÷ Hi-Lo game

You are playing a number guessing game with a friend. Your friend is thinking of a number between 1 and k, and you have to guess it. After every guess, your friend will say whether your guess is too high, too low, or correct. Prove that you can always guess the number in $\lceil \log_2(k) \rceil$ guesses.

This follows from choosing $\frac{k}{2}$ and then picking the middle element of this smaller range. This would allow us to find the number in $\lceil \log_2(k) \rceil$ queries.

This idea also works for slightly less direct questions:

X Hamburgers (Codeforces 371C)

Polycarpus have a fixed hamburger recipe using B pieces of bread , S pieces of sausage and C pieces of cheese; per burger.

At the current moment, in his pantry he has:

- n_b units of bread,
- n_s units of sausage,
- n_c units of cheese.

And the market prices per unit is:

- p_b rubles per bread,
- p_s rubles per sausage,
- p_c rubles per cheese.

Polycarpus's wallet has r rubles.

Each hamburger must be made exactly according to the recipe (ingredients cannot be split or substituted), and the store has an unlimited supply of each ingredient.

Write function `burgers :: (Int, Int, Int) → (Int, Int, Int) → (Int, Int, Int) → Int -> Int` which takes (B, S, C), (n_b, n_s, n_c), (p_b, p_s, p_c) and r and tells us how many burgers can Polycarpus make.

Examples

```
burgers (3,2,1) (6,4,1) (1,2,3)              4  = 2
burgers (2,0,1) (1,10,1) (1,10,1)           21  = 7
burgers (1,1,1) (1,1,1) (1,1,3)    1000000000000  = 200000000001
```

This question may look like a combinatorics or recursion question, but any of those approaches will be very inefficient.

Let's try to algebraically compute how much money is needed to make x burgers. We can define this cost function as cost times the number of ingredient required minus the amount already in pantry. This will something like:

$$f(x) = p_b \max(0, x \cdot B - n_b) + p_s \max(0, x \cdot S - n_s) + p_c \max(0, x \cdot C - n_c)$$

And now we want to look for maximal x such that $f(x) \leq r$. Well, that can be done using Binary search!

```
burgers (b, s, c) (nb, ns, nc) (pb, ps, pc) r = binarySearch 0 upperBound
  where
    -- Cost function f(x)
    cost x = let needB = max 0 (x * b - nb)
                 needS = max 0 (x * s - ns)
                 needC = max 0 (x * c - nc)
             in  needB *  pb +
                 needS *  ps +
                 needC *  pc

    upperBound = maximum [b,s,c] + r

    binarySearch low high
      | low > high = high
      | otherwise  =
         let mid = (low + high) `div` 2
         in if cost mid <= r
               then binarySearch (mid + 1) high
               else binarySearch low (mid - 1)
```

Here ia a similar exercise for your practice.

X House of Cards (Codeforces 471C)

- A house of cards consists of some non-zero number of floors.
- Each floor consists of a non-zero number of rooms and the ceiling. A room is two cards that are leaned towards each other. The rooms are made in a row, each two adjoining rooms share a ceiling made by another card.
- Each floor besides for the lowest one should contain less rooms than the floor below.

Please note that the house may end by the floor with more than one room, and in this case they also must be covered by the ceiling. Also, the number of rooms on the adjoining floors doesn't have to differ by one, the difference may be more.

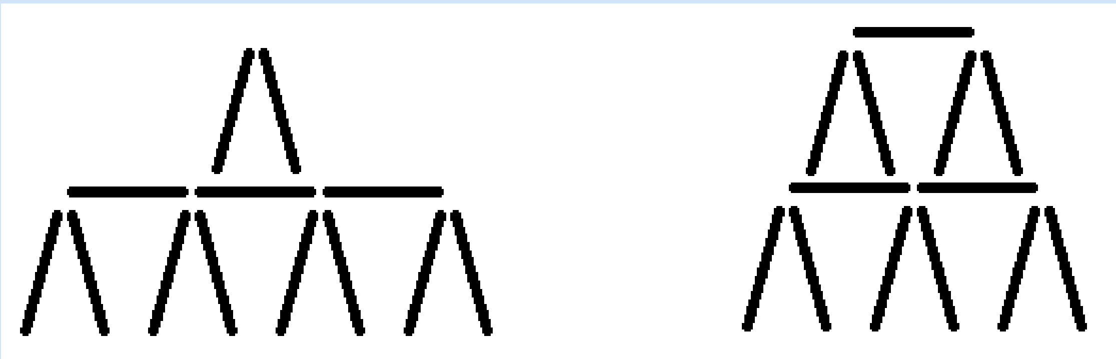
The height of the house is the number of floors in it.

Given n cards, it is possible that you can make a lot of different houses of different heights. Write a function `houses :: Integer → Integer` to count the number of the distinct heights of the houses that they can make using exactly n cards.

Examples

```
count 13 = 1
count 6 = 0
```

In the first sample you can build only these two houses (remember, you must use all the cards):



Thus, 13 cards are enough only for two floor houses, so the answer is 1.

The six cards in the second sample are not enough to build any house.

The reason we are interested in this methodology is as we could do this to find roots of polynomials, especially roots. How?

While using a raw binary 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 tolerance. Here is what it looks like:

λ **Square root by binary search**

```
bsSqrt :: Float → Float → Float
bsSqrt n tolerance
  | 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
```

Cube Root

Write a function `bsCbrt :: Float → Float → Float` which calculates the cube root of a number upto some tolerance using binary search.

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 exercise.

§3.12.2. Taylor Series

We know that $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots$. For small $x, \ln(1+x) \approx x$. So if we can create a scheme to make x small enough, we could get the logarithm 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 program heavier) or use more sophisticated mathematical methods which while more accurate would slow the program down. There is an exercise 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 neighborhood 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, similar 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)`.

X Inverse Trig

Use taylor approximation and trigonometric identities to define inverse sin(`asin`), inverse cos(`acos`) and inverse tan(`atan`).

§3.13. Exercises

X Collatz

Collatz conjecture 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 function `col :: Integer → Integer` which, given a n, finds the smallest 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 initial 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 intuition 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 absolutely greater than 1. Otherwise, 1 or -1 is a good guess. We leave it to your mathematical intuition 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 arithmetic-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 recursively 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 $x 2^m > 2^{\frac{p}{2}}$.

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 logarithm 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 getting a better `m` is not hard, this is just simpler.

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`?

Modular Exponentiation

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

X Bean Nim (Putnam 1995, B5)

A game starts with four heaps of beans containing a, b, c, and d beans. A move consists of taking either

(a) one bean from a heap, provided at least two beans are left behind in that heap, or

(b) a complete heap of two or three beans.

The player who takes the last heap wins. Do you want to go first or second?

Write a recursive function to solve this by brute force. Call it `naiveBeans :: Int → Int → Int → Int → Bool` which gives `True` if it is better to go first and `False` otherwise. Play around with this and make some observations.

Now write a much more efficient (should be one line and has no recursion) function `smartBeans :: Int → Int → Int → Int → Bool` which does the same.

X Squares and Rectangles on a chess grid

Write a function `squareCount :: Int → Int` to count number of squares on a $n \times n$ grid. For example, `squareCount 1 = 1` and `squareCount 2 = 5` as four 1×1 squares and one 2x2 square.

Furthermore, also make a function `rectCount :: Int → Int` to count the number of rectangles on a $n \times n$ grid.

Finally, make `genSquareCount :: (Int, Int) → Int` and `genRectCount :: (Int, Int) → Int` to count number of squares and rectangle in a $a \times b$ grid.

X Knitting Baltik (COMPFEST 13, Codeforces 1575K)

Mr. Chanek wants to knit a batik, a traditional cloth from Indonesia. The cloth forms a grid with size $m \times n$. There are k colors, and each cell in the grid can be one of the k colors.

Define a sub-rectangle as an pair of two cells $((x_1, y_1), (x_2, y_2))$, denoting the top-left cell and bottom-right cell (inclusively) of a sub-rectangle in the grid. Two sub-rectangles $((x_1, y_1), (x_2, y_2))$ and $((x_1, y_1), (x_2, y_2))$ have the same pattern if and only if the following holds:

(i) they have the same width ($x_2 - x_1 = x_4 - x_3$);

(ii) they have the same height ($y_2 - y_1 = y_4 - y_3$);

(iii) for every pair i, j such that $0 \leq i \leq x_2 - x_1$ and $0 \leq j \leq y_2 - y_1$, the color of cells $(x_1 + i, y_1 + j)$ and $(x_3 + i, y_3 + j)$ is the same.

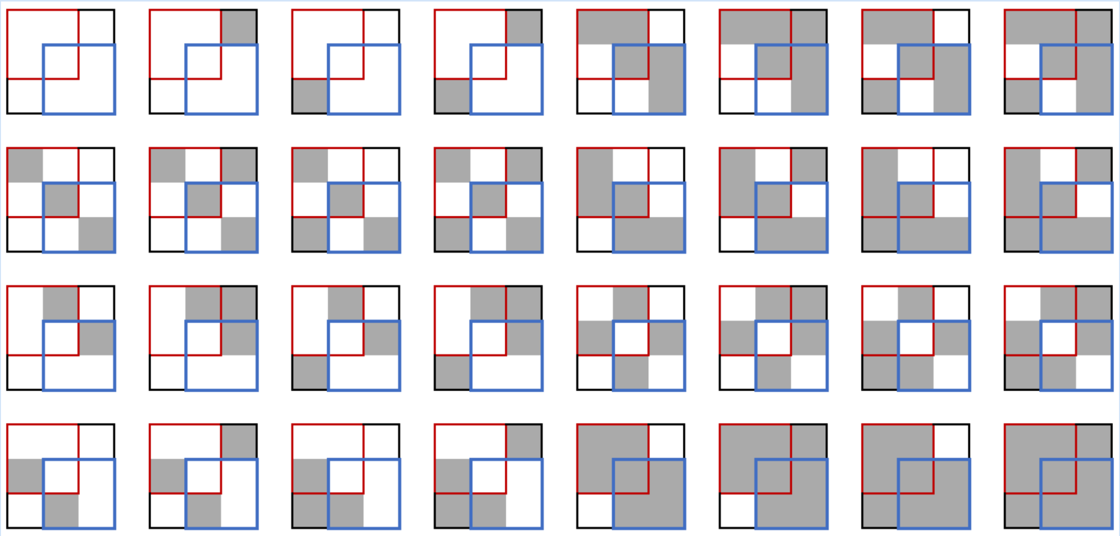
Write a function `countBaltik` of type `(Int, Int) → Int → (Int, Int) → (Int, Int) → (Int, Int) → Integer` to count the number of possible batik color combinations, such that the subrectangles $((a_x, a_y), (a_x + r - 1, a_y + c - 1))$ and $((b_x, b_y), (b_x + r - 1, b_y + c - 1))$ have the same pattern.

Input `countBaltik` takes as input:

- The size of grid (m, n)
- Number of colors k
- Size of sub-rectangle (r, c)
- (a_x, a_y)
- (b_x, b_y)

and should output an integer denoting the number of possible batik color combinations.

For example: `countBaltik (3,3) 2 (2,2) (1,1) (2,2) = 32`. The following are all 32 possible color combinations in the first example.



X Modulo Inverse

Given a prime modulus $p > a$, according to Euclidean Division $p = ka + r$ where

$$\begin{aligned} ka + r &\equiv 0 \bmod p \\ &\Rightarrow ka \equiv -r \bmod p \\ &\Rightarrow -ra^{-1} \equiv k \bmod p \\ &\Rightarrow a^{-1} \equiv -kr^{-1} \bmod p \end{aligned}$$

Using this, implement `modInv :: Int → Int → Int` which takes in a and p and gives $a^{-1} \bmod p$.

Note that this reasoning does not hold if p is not prime, since the existence of a^{-1} does not imply the existence of r^{-1} in the general case.

X New Bakery(Codeforces)

Bob decided to open a bakery. On the opening day, he baked n buns that he can sell. The usual price of a bun is a coins, but to attract customers, Bob organized the following promotion:

- Bob chooses some integer $k(0 \leq k \leq \min(n, b))$.
- Bob sells the first k buns at a modified price. In this case, the price of the i-th $(1 \leq i \leq k)$ sold bun is $(b - i + 1)$ coins.
- The remaining $(n - k)$ buns are sold at a coins each.

Note that k can be equal to 0. In this case, Bob will sell all the buns at a coins each.

Write a function `profit :: Int → Int → Int → Int` Help Bob determine the maximum profit he can obtain by selling all n buns with a being the normal price and b the price of first bun to be sold at a modified price.

Example

```
profit          4          4           5  = 17
profit          5          5           9  = 35
profit         10         10           5  = 100
profit 1000000000 1000000000 1000000000  = 1000000000000000000
profit 1000000000 1000000000          1  = 1000000000000000000
profit       1000          1        1000  = 500500
```

Note

In the first test case, it is optimal for Bob to choose $k = 1$. Then he will sell one bun for 5 coins, and three buns at the usual price for 4 coins each. Then the profit will be $5 + 4 + 4 + 4 = 17$ coins.

In the second test case, it is optimal for Bob to choose $k = 5$. Then he will sell all the buns at the modified price and obtain a profit of $9 + 8 + 7 + 6 + 5 = 35$ coins.

In the third test case, it is optimal for Bob to choose $k = 0$. Then he will sell all the buns at the usual price and obtain a profit of $10 \cdot 10 = 100$ coins.

x Sumac

A Sumac sequence starts with two non-zero integers t_1 and t_2.

The next term, $t_3 = t_1 - t_2$

More generally, $t_n = t_{n-2} - t_{n-1}$

The sequence ends when $t_n \leq 0$. All values in the sequence must be positive.

Write a function `sumac :: Int → Int → Int` to compute the length of a Sumac sequence given the initial two terms, t_1 and t_2.

Examples(Sequence is included for clarification)

```
(t1,t2)   Sequence           n
--------------------------------
(120,71)  [120,71,49,22,27] 5
(101,42)  [101,42,59]       3
(500,499) [500,499,1,498]   4
(387,1)   [387,1,386]       3
(3,-128)  [3]               1
(-2,3)    []                0
(3,2)     [3,2,1,1]         4
```

x Binet Formula

Binet's formula is an explicit, closed form formula used to find the nth term of the Fibonacci sequence. It is so named because it was derived by mathematician Jacques Philippe Marie Binet, though it was already known by Abraham de Moivre.

Thr problem with this remarkable formula is that it is cluttered with irrational numbers, specifically $\sqrt{5}$.

$$F_n = \frac{\left(1 + \sqrt{5}\right)^n - \left(1 - \sqrt{5}\right)^n}{2^n \sqrt{5}}$$

While computing using the Binet formula would only take $2 * \log(n) + 2$ operations (exponentiation takes $\log(n)$ time), doing so directly is out of the question as we can't represent $\sqrt{5}$ exactly and the small errors in the approximation will accumulate due to the number of operations.

So an idea is to do all computations on a tuple (a, b) which represents $a + b\sqrt{5}$, We will need to define **addition, subtraction, multiplication and division** on these tuples as well as define a **fast exponentiation** here.

With that in hand, Write a function `fibMod :: Integer → Integer` which computes Fibonacci numbers using this method.

X A puzzle (UVA 10025)

A classic puzzle involves replacing each ? with one can set operators + or −, in order to obtain a given k

$$? \, 1 ? \, 2 ? \dots \; ? \, n = k$$

For example: to obtain $k = 12$, the expression to be used will be:

$$-1 + 2 + 3 + 4 + 5 + 6 - 7 = 12$$

with $n = 7$

Write function `puzzleCount :: Int → Int` which given a k tells us the smallest n such that the puzzle can be solved.

Examples

```
puzzleCount 12 = 7
puzzleCount -3646397 = 2701
```

Rating Recalculation (Code Forces)

It is well known in the Chess Federation that the boundary for the Grandmaster title is carefully maintained just above the rating of International Master Wupendra Wulkarni. However, due to a recent miscalculation in the federation's new rating system, Wulkarni was mistakenly awarded the Grandmaster title.

To correct this issue, the federation has decided to revamp the division system, ensuring that Wupendra is placed into Division 2 (International Master), well below Grandmaster status.

A simple rule like `if rating ⩽ wupendraRating then div = max div 2` would be too obvious and controversial. Instead, the head of the system, Mike, proposes a more subtle and mathematically elegant solution.

First, Mike chooses the integer parameter $k \geq 0$.

Then, he calculates the value of the function $f(r-k, r)$, where r is the user's rating, and

$$f(n, x) := \frac{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \ + \frac{x^n}{n!}}{e^x}$$

Finally, the user's division is defined by the formula

$$\operatorname{div}(r) = \left\lfloor \frac{1}{f(r-k, r)} \right\rfloor - 1$$

.

Write function `ratingCon :: Int → Int` to find the minimum k, given Wupendra's rating, so that the described algorithm assigns him a division strictly greater than 1 and GM Wulkarni doesn't become a reality.

Examples

```
ratingCon 5 = 2
ratingCon 100 = 5
ratingCon 200 = 7
ratingCon 2500 = 23
ratingCon 3000 = 25
ratingCon 3500 = 27
```

Knuth's Arrow

Knuth introduced the following notation for a family of math notation:

$$3 \cdot 4 = 12$$
$$3 \uparrow 4 = 3 \cdot (3 \cdot (3 \cdot 3)) = 3^4 = 81$$
$$3 \uparrow\uparrow 4 = 3 \uparrow (3 \uparrow (3 \uparrow 3)) = 3^{3^{3^3}} = 3^{7625597484987}$$
$$\approx 1.258014290627491317860390698203281215518046714316596015189 67 \times 10^{3638334640024}$$
$$3 \uparrow\uparrow\uparrow 4 = 3 \uparrow\uparrow (3 \uparrow\uparrow (3 \uparrow\uparrow 3))$$

You can see the pattern as well as the extreme growth rate. Make a function `knuthArrow :: Integer → Int → Integer → Integer` which takes the first argument, number of arrows and second argument and provides the answer.

X Caves (IOI 2013, P4)

While lost on the long walk from the college to the UQ Centre, you have stumbled across the entrance to a secret cave system running deep under the university. The entrance is blocked by a security system consisting of N consecutive doors, each door behind the previous; and N switches, with each switch connected to a different door.

The doors are numbered $0, 1, \cdots, 4999$ in order, with door 0 being closest to you. The switches are also numbered $0, 1, \cdots, 4999$, though you do not know which switch is connected to which door.

The switches are all located at the entrance to the cave. Each switch can either be in an up or down position. Only one of these positions is correct for each switch. If a switch is in the correct position then the door it is connected to will be open, and if the switch is in the incorrect position then the door it is connected to will be closed. The correct position may be different for different switches, and you do not know which positions are the correct ones.

You would like to understand this security system. To do this, you can set the switches to any combination, and then walk into the cave to see which is the first closed door. Doors are not transparent: once you encounter the first closed door, you cannot see any of the doors behind it. You have time to try $70,000$ combinations of switches, but no more. Your task is to determine the correct position for each switch, and also which door each switch is connected to.

X Carnivel (CEIO 2014)

Each of Peter's N friends (numbered from 1 to N) bought exactly one carnival costume in order to wear it at this year's carnival parties. There are C different kinds of costumes,numbered from 1 to C. Some of Peter's friends, however, might have bought the same kind of costume. Peter would like to know which of his friends bought the same costume. For this purpose, he organizes some parties, to each of which he invites some of his friends.

Peter knows that on the morning after each party he will not be able to recall which costumes he will have seen the night before, but only how many different kinds of costumes he will have seen at the party. Peter wonders if he can nevertheless choose the guests of each party such that he will know in the end, which of his friends had the same kind of costume. Help Peter!

Peter has $N \leq 60$ friends and we can not have more than 365 parties(as we want to know the costumes by the end of the year).

Types as Sets

§4.1. 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 **singleton set** is a ***set that contains exactly one element***, such as $\{34\}$, $\{\triangle\}$, the set of natural numbers strictly between 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?

§4.2. Types

We have encountered 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 boolean 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 -

§4.2.1. `::` 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`.

§4.2.2. `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?

§4.2.3. `( 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')`, then it would tell me that the value's type is `(Bool, Char)` -

λ **type of a pair**
```
>>> :type (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?

§4.2.4. `()` 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` -

λ elements of unit type
```
>>> listOfAllElements :: [()]
[()]
```

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

function to unit

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

function from unit

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

§4.2.5. 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.

Therefore, no intersection operation is defined for types.

§4.2.6. 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**.

subtype

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

§4.2.7. 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 alternative, i.e, the "10" here comes from $\mathbb{N}$, not $\mathbb{R}$.

§4.2.8. `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 following 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 `Char` (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?

§4.2.9. 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 to 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 `inverseOfReciprocal` s.t.

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

as follows -

λ function from a maybe type

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

§4.2.10. `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`.

§4.3. Currying

Let's try to explore some more elaborate types.
For example, let us try to find out the **type of the derivative operator**,

$$\frac{d}{dx}$$

Let $\mathbb{D}$ be the set of all differentiable $\mathbb{R} \to \mathbb{R}$ functions.

Now, for any $f \in \mathbb{D}$, i.e., for any differentiable function $f : \mathbb{R} \to \mathbb{R}$,
we know that $\frac{df}{dx}$ will be also be a $\mathbb{R} \to \mathbb{R}$ function.

Specifically, we could define

$$\frac{df}{dx} := \left(p \mapsto \lim_{h \to 0} \frac{f(p+h) - f(p)}{h}\right)$$

Therefore, the function $\frac{d}{dx}$
takes an input f of type $\mathbb{D}$,
and produces an output $\frac{df}{dx}$ of type $\mathbb{R} \to \mathbb{R}$, which is written set-theoretically as $\mathbb{R}^{\mathbb{R}}$.

An thus we obtain the type of the derivative operator as

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

or more formally,

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

But we know another syntax for writing the derivative, which is -

$$\left.\frac{df}{dx}\right|_p$$

, which refers to the derivative evaluated at a point $p \in \mathbb{R}$.

Here, the definition could be written as

$$\left.\frac{df}{dx}\right|_p := \lim_{h\to 0}\frac{f(p+h)-f(p)}{h}$$

So here there are two inputs, namely $f \in \mathbb{D}$ and $p \in \mathbb{R}$,
and an output $\left.\frac{df}{dx}\right|_p$, which is of type $\mathbb{R}$.

That leads us to the type -

$$\frac{d}{dx} : \mathbb{D} \times \mathbb{R} \to \mathbb{R}$$

We understand that these two definitions are equivalent.
So now the question is, which type do we use?

High-school math usually chooses to use the $\mathbb{D} \times \mathbb{R} \to \mathbb{R}$ style of typing.

Haskell, and in several situations math as well,
defaults to the $\mathbb{D} \to (\mathbb{R} \to \mathbb{R})$, or equivalently $\mathbb{D} \to \mathbb{R}^{\mathbb{R}}$ style of typing.

In general, that means that if a function $F : A \to (B \to C)$

takes an input from A,
and gives as output a $B \to C$ function,

then it is equivalent to saying $F : A \times B \to C$, which would make F a function

that takes inputs of type A and B respectively,
and gives an output of type C.

[5]

We have just seen the example where F was $\frac{d}{dx}$ and A, B, C were $\mathbb{D}, \mathbb{R}, \mathbb{R}$ respectively.

However, this has more profound consequences than what appears at first glance, in Haskell as well as in post-high-school mathematics.

This is due to looking in the opposite direction, i.e., taking a definition like

$$\left.\frac{df}{dx}\right|_p := \lim_{h\to 0}\frac{f(p+h)-f(p)}{h}$$

and rephrasing it as

$$\frac{df}{dx} := \left(p \mapsto \lim_{h\to 0}\frac{f(p+h)-f(p)}{h}\right)$$

[5]will be proven soon

[6]will be proven soon

In general, if a function $F : A \times B \to C$

takes inputs of type A and B respectively,
and gives an output of type C.

then it is equivalent to saying $F : A \to (B \to C)$, which would make F a function

that takes an input from A,
and gives as output a $B \to C$ function.

6

This rephrasing is known as "currying".

§4.3.1. In Haskell

Again, for example,
if we have a function such as $\frac{d}{dx}$ which has **2 inputs** (f and p),
we can use it by **only giving the first input**
in the following sense -

$$\frac{df}{dx} := \left(p \mapsto \left.\frac{df}{dx}\right|_p\right)$$

Let's see how it works in Haskell.

≑ **currying rule**

If we have a function `f` that takes 2 inputs (say `x` and `y`), then we can use `f x` as

```
f x = \ y → f x y
```

We know that `(+)` is a function that takes in two `Integer`s and outputs an `Integer`.
This means that A, B, C are `Integer`, `Integer`, `Integer` respectively.

By currying or rephrasing, this would mean that we could treat `(+)` like a function that takes a single input of type `Integer` (i.e., A) and outputs a function of type `Integer → Integer` (i.e., $B \to C$).

In fact, that's exactly what Haskell lets you do -

```
>>> :type +d (+) 17
(+) 17 :: Integer → Integer
```

Meaning that when `(+)` is given the `Integer` input `17`, it outputs the function `(+) 17`, of type `Integer → Integer`.

More explicitly, by the ≑ **currying rule**, we have that

```
(+) 17 = \ y → (+) 17 y
```

Thus, what does this function `(+) 17` actually do?
Simple! It is a function that takes in any `Integer` and adds `17` to it.

So, for example,
If we define -

λ **currying usage**
```
test = (+) 17
```

it behaves as such -

```
>>> test 0
17
>>> test 1
18
>>> test 12
29
>>> test (-17)
0
```

Another -

```
>>> :type +d (*)
(*) :: Integer -> Integer -> Integer

>>> :type +d (*) 2
(*) 2 :: Integer -> Integer
```

Meaning that when `(*)` is given the `Integer` input `2`, it outputs the function `(*) 2`, of type `Integer -> Integer`.

More explicitly, by the ÷ **currying rule**, we have that

```
(*) 2 = \ y -> (*) 2 y
```

Thus, the function `(*) 2`
takes in an `Integer` input
and multiplies it by `2`, i.e., doubles it.

So if we define -

λ **another currying usage**
```
doubling :: Integer -> Integer
doubling = (*) 2
```

it behaves as such -

```
>>> doubling 0
0
>>> doubling 1
2
>>> doubling 12
24
>>> doubling (-17)
-34
```

§4.3.2. Understanding through Associativity

§4.3.2.1. Of `→`

The ≑ **currying rule** essentially allows us to view a function of type `A → B → C` as of type `A → ( B → C )`.

This is due to the fact that as an ≑ **infix binary operator**, the `→` operator is ≑ **right-associative**.

Recalling the definition of ≑ **right-associative**, this means that,
for any `X`, `Y`, `Z` -

```
X → Y → Z
```

is actually equivalent to

```
X → ( Y → Z )
```

And thus the ≑ **currying rule** is justified.

§4.3.2.2. Of Function Application

Let us take the ≑ **currying rule**

```
f x = \ y → f x y
```

Applying a few transformations to both sides -

```
( f x ) == ( \ y → f x y )
-- applying both sides to y
( f x ) y == ( \ y → f x y ) y
-- simplifying
( f x ) y == f x y
-- exchanging LHS and RHS
f x y == ( f x ) y
```

Thus we obtain the result that any time we write

```
f x y
```

it is actually equivalent to

```
( f x ) y
```

This means that "function application" is ≑ **left-associative**. (Recall the definition of ≑ **left-associative** and see if this makes sense)

That is, if we apply a function `f` to 2 inputs `x` and `y` in the form `f x y`,
then `f x` (the application on the **left**) is evaluated first (as seen in `( f x ) y`) and then the obtained `( f x )` is applied on `y`.

§4.3.2.3. Operator Currying Rule

We have already seen the ≑ **currying rule**. However it can be extended in a special way when the function is an ≑ **infix binary operator**.

≑ **operator currying rule**

If we have an ≑ **infix binary operator** `?`, then we can assume the following due to the ≑ **currying rule** -

```
(?) x = \ y → (?) x y -- the normal currying rule
-- which is equivalent to
(?) x = \ y → x ? y
```

But we may further assume

```
(x?) = \ y → x ? y
```

and also

```
(?y) = \ x → x ? y
```

This means that while the ≑ **currying rule** allowed us to give only the ***first input*** (i.e., `x`) and get a meaningful function out of it,
the **operator currying rule** further allows to do something similar by only giving the ***second input*** (i.e., `y`).

For example, -

```
>>> :type +d (^)
(^) :: Integer → Integer → Integer

>>> :type +d (^2)
(^2) :: Integer → Integer
```

Meaning that when the ≑ **infix binary operator** `^` is given the `Integer` input `2` in place of its second input, it outputs the function `(^2)`, of type `Integer → Integer`.

More explicitly, by the ≑ **operator currying rule** -

```
(^2) = \ x → x ^ 2
```

Thus, `(^2)` is a function that takes an `Integer` `x` and raises to to the power of `2`, i.e., **squares** it.

So if we define -

λ operator currying usage

```
squaring :: Integer → Integer
squaring = (^2)
```

it will show the following behaviour -

```
>>> squaring 0
0
>>> squaring 1
1
>>> squaring 12
144
>>> squaring (-17)
289
```

For another example, we can define

λ another operator currying usage

```
cubing :: Integer → Integer
cubing = (^3)
```

which works quite similarly.

§4.3.3. Proof of the Currying Theorem

What follows is an **OPTIONAL** formal rigorous proof of the following statement -

In general, if a function $F : A \times B \to C$

takes inputs of type A and B respectively,
and gives an output of type C.

then it is equivalent to saying $F : A \to (B \to C)$, which would make F a function

that takes an input from A,
and gives as output a $B \to C$ function.

What we are going to prove is that
there exists a bijection
from
the set $\{F \mid F : A \times B \to C\}$
to
the set $\{G \mid G : A \to (B \to C), \text{or equivalently}, G : A \to C^B\}$

Note that the set $\{F \mid F : A \times B \to C\}$ can be expressed as $C^{A \times B}$
and that the set $\{G \mid G : A \to C^B\}$ can be expressed as $\left(C^B\right)^A$

Therefore, we have to prove there exists a bijection : $C^{A \times B} \to \left(C^B\right)^A$

x **finite currying**

Is there an easy way to prove the theorem in the case that A, B, C are all finite sets?

Theorem $\exists$ a bijection : $C^{A \times B} \to (C^B)^A$

Proof Define the function $\mathcal{C}$ as follows -

$$\begin{aligned} \mathcal{C} &: C^{A \times B} \to (C^B)^A \\ \mathcal{C}(F) &:= G \\ &\quad \text{where} \\ &\quad G : A \to C^B \\ &\quad G(a) := (b \mapsto F(a,b)) \end{aligned}$$

If we can prove that $\mathcal{C}$ is bijective, we are done!
In order to do that, we will prove that $\mathcal{C}$ is injective as well as a surjective.

Claim : $\mathcal{C}$ is injective
Proof :

$$\begin{aligned} & & \mathcal{C}(F_1) &== \mathcal{C}(F_2) \\ \Rightarrow & & G_1 &== G_2 \quad \text{, where } G_1(a) := (b \mapsto F_1(a,b)) \text{ and } G_2(a) := (b \mapsto F_2(a,b)) \\ \Rightarrow & \forall a \in A, & G_1(a) &== G_2(a), \text{ where } G_1(a) := (b \mapsto F_1(a,b)) \text{ and } G_2(a) := (b \mapsto F_2(a,b)) \\ \Rightarrow & \forall a \in A, & (b \mapsto F_1(a,b)) &== (b \mapsto F_2(a,b)) \\ \Rightarrow & \forall a \in A, \forall b \in B, & (b \mapsto F_1(a,b))(b) &== (b \mapsto F_2(a,b))(b) \\ \Rightarrow & \forall a \in A, \forall b \in B, & F_1(a,b) &== F_2(a,b) \\ \Rightarrow & \forall p \in A \times B, & F_1(p) &== F_2(p) \\ \Rightarrow & & F_1 &== F_2 \end{aligned}$$

Claim : $\mathcal{C}$ is surjective
Proof : Take an arbitrary $H \in (C^B)^A$.
In other words, take an arbitrary function $H : A \to (B \to C)$.
Define a function J as follows -

$$\begin{aligned} J &: A \times B \to C \\ J(a,b) &:= (H(a))(b) \end{aligned}$$

Now,

$$\begin{aligned} \mathcal{C}(J) :=\ & G \text{ , where } G(a) := (b \mapsto J(a,b)) \\ ==\ & G \text{ , where } G(a) := (b \mapsto (H(a))(b)) \\ ==\ & G \text{ , where } G(a) := (H(a)) \qquad [\because (x \mapsto f(x)) \text{ is equivalent to just } f\ ] \\ ==\ & G \text{ , where } G == H \qquad [\because f(x) := g(x) \text{ means that } f == g\ ] \\ ==\ & H \end{aligned}$$

That means we have proved that

$$\forall H \in (C^B)^A, \exists J \text{ such that } \mathcal{C}(J) == H$$

Therefore, by the definition of surjectivity, we have proven that $\mathcal{C}$ is surjective.
As a result, we are done with the overall proof as well!

∎

§4.4. Exercises

Symmetric Difference

(i) Define the symmetric difference of the sets A and B as $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

Prove that this is a commutative and associative operation.

(ii) The set $A_1 \Delta A_2 \Delta ... \Delta A_n$ consists of those elements that belong to an odd number of the A_i's.

Set of Size

(i) Give a type with exactly 32 elements. (ii) Give a type with exactly 108 elements. (iii) Give a type with exactly 19 elements.

Unions and Intersections

Prove: (i)

$$\forall i \in I, x_i \subseteq y \Rightarrow \bigcup_{i \in I} x_i \in y$$

(ii)

$$\forall i \in I, y \subseteq y \Rightarrow y \in \bigcap_{i \in I} x_i$$

(iii)

$$\bigcup_{i \in I} (x_i \cup y_i) = \left(\bigcup_{i \in I} x_i\right) \cup \left(\bigcup_{i \in I} y_i\right)$$

(iv)

$$\bigcap_{i \in I} (x_i \cap y_i) = \left(\bigcap_{i \in I} x_i\right) \cap \left(\bigcap_{i \in I} y_i\right)$$

(v)

$$\bigcup_{i \in I} (x_i \cap y) = \left(\bigcup_{i \in I} x_i\right) \cap y$$

(vi)

$$\bigcap_{i \in I} (x_i \cup y) = \left(\bigcap_{i \in I} x_i\right) \cup y$$

x Flavoured like Curry

$A \sim B$ means that there exists a bijection between A and B.

Prove:

(i)

$$A(B + C) \sim AB + AC$$

(ii) $(B \cup C)^A \sim B^A \times C^A$ provided $B \cap C = \emptyset$

(iii)

$$C^{A \times B} \sim C^A \times C^B$$

x Eckman-Hilton Argument

÷ Unital Operator

A binary operator $*$ over a set S is **Unital** if there exists $l, r \in S$ s.t. $\forall x \in S, l * x = x * r = x$. (You can also prove $r = l$! This is why we normally label this $l = r = 1_S$ in abstract algebra.)

If a set S has two unital operations $\star$ and $\cdot$ defined on it such that:

$$(w \cdot x) \star (y \cdot z) = (w \star x) \cdot (y \star z)$$

Prove that $\star \equiv \cdot$.

x Associative Operators

(i) How many different associative binary logical operators can be defined? (Formally, non-isomorphic associative binary logical operators. Informally, isomorphic means “same up to relabelling.”). Can you define all of them?

(ii) Suppose we have an associative binary operator on a set S of size $0 < k < \infty$. Prove that $\exists x \in S, x \cdot x = x$.

Note: A set with an associative binary operation is called a semi-group. The first question can also be posed for a set of general size, but we don’t know the answer or an algorithm to get the answer beyond sets of size 9.

Contents

X Feels Abstract

≑ Shelf

A set with an binary operation $\triangleright$ which left distributes over itself is called a left shelf, that is $\forall x, y, z \in S, x \triangleright (y \triangleright z) = (x \triangleright y) \triangleright (x \triangleright z)$.

A similar definition holds for right shelf and the symbol often used is $\triangleleft$.

(i) Prove that a unital left shelf is associative. In other words: if there exists $1_S \in S$ such that $1 \triangleright x = x \triangleright 1 = x$, then $\triangleright$ is associative.

≑ Rack

A rack is a set R with two operations $\triangleright$ and $\triangleleft$, such that R is a left shelf over $\triangleright$ and a right shelf over $\triangleleft$ satisfying $x \triangleright (y \triangleleft x) = (x \triangleright y) \triangleleft x = y$.

(ii) Prove that in a rack R, $\triangleright$ distributes over $\triangleleft$ and vice versa. That is, $\forall x, y, z \in R; x \triangleright (y \triangleleft z) = (x \triangleright y) \triangleleft (x \triangleright z)$.

(iii) We call an operator $\star$ **idempotent** if $x \star x = x$. If for a rack, $\triangleright$ is idempotent; prove that $\triangleleft$ is idempotent.

≑ Quandle

A Quandle is a rack with $\triangleright$ and $\triangleleft$ being idempotent.

(iv) We call an operator $\star$ **left involute** if $x \star (x \star y) = y$. Prove that an idempotent, left involutive, left shelf is a quandle (recall that the shelf only has one operator, we need to somehow suitably define the other operation. Maybe do problem (v) for a hint?)

≑ Kei

An involutive quandle is called a **Kei**

(v) Prove that the set of points on the real plane with $x \triangleright y$ being the reflection of y over x is a Kei.

Note: Shelves, Racks, Quandles and Kai's are slowly entering mainstream math as ways to work with operations on exotic sets like set of knots or set of colorings of primes etc. The theory of Kai in this regard was formalized very recently (2024) by Davis and Schlank in "Arithmetic Kei Theory". The main use is still Knot Theory, but we would not be surprised to see it used in a number theory proof. A reference for this, and a pre-req for Davis and Schlank, is "Quandles" by Mohamed Elhamdadi and Sam Nelson.

X Enumerating the Rationals

(i) Prove that $\mathbb{Z}$ is countable, that is there is a surjection from $\mathbb{N} \to \mathbb{Z}$.[7]

(ii) Now write functions : `natToInt :: Integer → Integer` and `intToNat :: Integer → Integer` which takes a natural number and gives the corresponding integer and vice versa.

(iii) Prove that $\mathbb{N} \times \mathbb{N}$ is countable, that is there is a bijection between $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. While other arguments exist, we are big fans of enumerating in the order $(0,0) \to (0,1) \to (1,0) \to (0,2) \to (1,1) \to (2,0) \to \ldots$. What is the pattern?

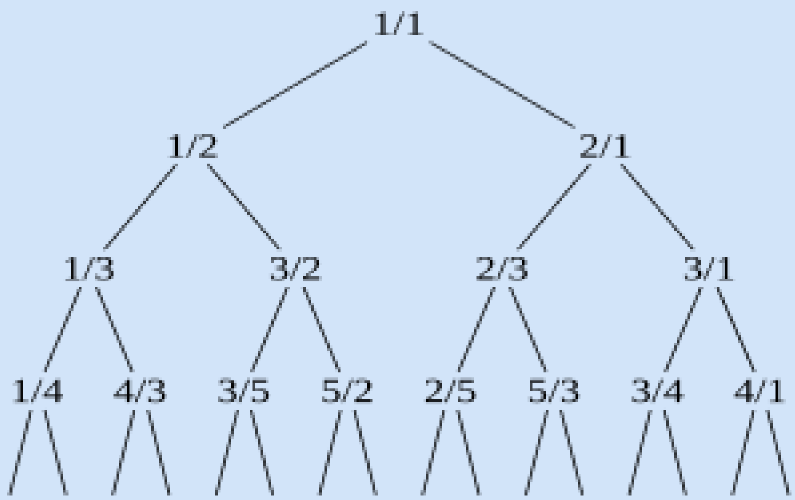
(iv) Now write a function `intToPair :: Int → (Int, Int)` which takes an integer and gives the corresponding rational and a function `pairToInt :: (Int, Int) → Int` which takes a pair and returns the corresponding integer.

X Calkin-Wilf Tree

Using the above exercise, we can see that $\mathbb{N} \times \mathbb{N}$ is 'larger'[8] than $\mathbb{Q}^+$, so we can claim that rationals are countable. We will attempt to prove that as well as enumerate the rationals.

(i) Prove that if $\frac{p}{q}$ is reduced, then $\frac{p+q}{q}$ and $\frac{p}{p+q}$ are reduced.

(ii) Prove that starting with $\frac{1}{1}$ and making the following tree by applying the above transformation will contain every rational:



(iii) Labeling the tree level by level, write a function `natToRat :: Int → (Int, Int)` which takes an natural number and gives the positive rational enumerates. (An approach could be to notice that we can represent the path one takes down the tree in binary)

(iv) Write a function `ratToNat :: (Int, Int) → Int` which takes a positive rational number and gives its position in the naturals.

[7]This may seem counterintuitive as the integers feel twice as large but they really are not

[8]It is not as you will see in a moment,

Introduction to Lists

A list is an ordered collection of objects, possibly with repetitions, denoted by

$$[\ \text{object}_0\ ,\ \text{object}_1\ ,\ \text{object}_2\ , \ldots ,\ \text{object}_{n-1}\ ,\ \text{object}_n\ ]$$

These objects are called the **elements of the list**.

In Haskell, the elements of a particular list all have to have the same type.

Thus, a list such as `[1,2,True,4]` is not allowed.

§5.1. Type of List

If the elements of a list each have type `T`, then the list is given the type `[T]`.

```
>>> :type +d [1,2,3]
[1,2,3] :: [Integer]

>>> :type +d ['a','Z','\STX']
['a','Z','\STX'] :: [Char]

>>> :type +d [True,False]
[True,False] :: [Bool]
```

§5.2. Creating Lists

There are several nice ways to create a list in Haskell.

§5.2.1. Empty List

The most basic approach is to create the empty list (a list containing no elements) by writing `[]`.

§5.2.2. Arithmetic Progression

Haskell has some luxurious syntax for declaring lists containing arithmetic progressions -

arithmetic progression syntax

```
>>> [1..6]
[1,2,3,4,5,6]

>>> [1,3..6]
[1,3,5]

>>> [1,-3.. -10]
[1,-3,-7]

>>> [0.5..4.9]
[0.5,1.5,2.5,3.5,4.5]
```

But, very usefully, it just doesn't work for numbers, but other types as well.

non-number arithmetic progressions

```
>>> [False ..True]
[False,True]

>>> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

§5.3. Functions on Lists

Now that we know how to create a list, how do we manipulate them into the data that we would want?

§5.3.1. List Comprehension

Well, the way we achieve this in sets is through **set comprehension**.

When we want the set of squares of the even natural numbers $\leq n$, we write -

$$\{m^2 \mid m \in \{0, 1, 2, 3, ..., n-1, n\}, 2 \text{ divides } m\}$$

Haskell lets us do the same with lists -

```
>>> n = 10
>>> [ m*m | m ← [0..n] , m `mod` 2 == 0 ]
[0,4,16,36,64,100]
```

When we want the set of pairs of numbers $\leq n$ whose highest common factor is 1, we write -

$$\{(x, y) \mid x, y \in \{0, 1, 2, 3, ..., n-1, n\}, \text{HCF}(x, y) == 1\}$$

,which can be expressed in haskell as

```
>>> n = 10
>>> [ (x,y) | x ← [1..n] , y ← [1..n] , gcd x y == 1 ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10),(2,1),(2,3),
(2,5),(2,7),(2,9),(3,1),(3,2),(3,4),(3,5),(3,7),(3,8),(3,10),(4,1),(4,3),
(4,5),(4,7),(4,9),(5,1),(5,2),(5,3),(5,4),(5,6),(5,7),(5,8),(5,9),(6,1),
(6,5),(6,7),(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,8),(7,9),(7,10),(8,1),
(8,3),(8,5),(8,7),(8,9),(9,1),(9,2),(9,4),(9,5),(9,7),(9,8),(9,10),(10,1),
(10,3),(10,7),(10,9)]
```

§5.3.2. Cons or `(:)`

The operator `:` (read as "cons") can be used to add a single element to the the beginning of a list.

```
>>> 5 : [8,2,3,0]
[5,8,2,3,0]

>>> 1 : [2,3,4]
[1,2,3,4]

>>> 7 : [10,2,35,92]
[7,10,2,35,92]

>>> True : [False,True,True,False]
[True,False,True,True,False]
```

However, the `:` operator is much more special than it appears, since -

- It can be used to pattern match lists
- It is how lists are defined in the first place

So, how can we use it for pattern matching?

λ pattern matching lists

```
>>> (x:xs) = [5,8,3,2,0]
>>> x
5
>>> xs
[8,3,2,0]
```

When we use the pattern `(x:xs)` to refer to a list, `x` refers to the first element of the list, and `xs` refers to the list containing the rest of the elements.

§5.3.3. Length

One of the most basic questions we could ask about lists is the number of elements they contain. The `length` function gives us that answers, counting repetitions as separate.

```
>>> length [5,5,5,5,5,5]
6

>>> length [5,8,3,2,0]
5

>>> length [7,10,2,35,92]
5

>>> length [False,True,True,False]
4
```

Ans we can use pattern matching to define it -

λ length of list

```
length []     = 0
length (x:xs) = 1 + length xs
```

This reads - " If the list is empty, then `length` is `0`.

If the list has a first element `x`, then the `length` is `1 + length of the list of the rest of the elements`. "

§5.3.4. Concatenate or `(++)`

The `++` (read as "concatenate") operator can be used to join two lists together.

```
>>> [5,8,2,3,0] ++ [122,32,44]
[5,8,2,3,0,122,32,44]

>>>  [False,True,True,False] ++ [True,False,True]
[False,True,True,False,True,False,True]
```

Again, we can define it by using pattern matching

λ concatenation of lists

```
[]     ++ ys = ys
(x:xs) ++ ys = x : ( xs ++ ys )
```

This reads - " Suppose we are concatenating a list to the front of the list `ys`.

If the list is empty, then of course the answer is just `ys`.

If the list has a first element `x`, and the rest of the elements form a list `xs`, then we can first concatenate `xs` and `ys`, and then add `x` at the beginning of the resulting list. "

§5.3.5. Head and Tail

The `head` function gives the first element of a list.

```
>>> head [5,8,3,2,0]
5

>>> head [7,10,2,35,92]
7

>>> head [False,True,True,False]
False
```

And it can be defined using pattern-matching -

λ head of list

```
head (x:xs) = x
```

The `tail` function provides the rest of the list after the first element.

```
>>> tail [5,8,3,2,0]
[8,3,2,0]

>>> tail [7,10,2,35,92]
[10,2,35,92]

>>> tail [False,True,True,False]
[True,True,False]
```

And it can be defined using pattern-matching -

λ **tail of list**

```
tail (x:xs) = xs
```

But how are these functions supposed to work if there is no first element at all, such as in the case of `[]`? They produce errors when applied to the empty list! -

```
>>> head []
*** Exception: Prelude.head: empty list
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\List.hs:1644:3 in base:GHC.List
  errorEmptyList, called at libraries\base\GHC\List.hs:87:11 in
base:GHC.List
  badHead, called at libraries\base\GHC\List.hs:83:28 in base:GHC.List
  head, called at <interactive>:6:1 in interactive:Ghci6
```

```
>>> tail []
*** Exception: Prelude.tail: empty list
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\List.hs:1644:3 in base:GHC.List
  errorEmptyList, called at libraries\base\GHC\List.hs:130:28 in
base:GHC.List
  tail, called at <interactive>:7:1 in interactive:Ghci6
```

Note that, in our definitions, we have not handled the case of the input being `[]`!

So, it is advised to use the function `uncons` from `Data.List`, which adopts the philosophy we saw in λ **function to a maybe type**, which is

if the function gives an error, output Nothing instead of the error

Thus, for non-empty `l`, `uncons l` returns `Just (head l, tail l)`,
and when `l` is empty, `uncons l` returns `Nothing`.

Let's test this in GHCi -

```
>>> import Data.List
>>> uncons [5,8,3,2,0]
Just (5,[8,3,2,0])
>>> uncons []
Nothing
```

And the definition -

λ **uncons of list**

```
uncons []     = Nothing
uncons (x:xs) = Just ( x , xs )
```

Also consider the functions `safeHead` and `safeTail` from `Distribution.Simple.Utils`.

§5.3.6. Take and Drop

There are some "generalized" functions corresponding to `head` and `tail`, namely `take` and `drop`,

`take n l` gives the first `n` elements of `l`.

```
>>> take 3 [5,8,3,2,0]
[5,8,3]

>>> take 4 [7,10,2,35,92]
[7,10,2,35]

>>> take 2 [False,True,True,False]
[False,True]
```

And the definition -

λ take from list

```
take 0    l      = []
take n (x:xs) = x : take (n-1) xs
take n []      = []
```

This reads - “ If we `take` only `0` elements, the result will of course be the empty list `[]`.

If we want to take `n` elements, then we can take the first element and then the first `n-1` elements from the rest.
But why the last line of the definition? “ The last line of the function may look strange, but -

X Exercise

Explain why, without the last line of the definition, the function might give an unexpected error.

`drop n l` gives `l`, excluding the first `n` elements.

```
>>> drop 3 [5,8,3,2,0]
[2,0]

>>> drop 4 [7,10,2,35,92]
[92]

>>> drop 2 [False,True,True,False]
[True,False]
```

And the definition -

λ drop from list

```
drop 0    l      = l
drop n (x:xs) = drop (n-1) xs
drop n []      = []
```

X Exercise

Prove that the above definition works as told in the description of the functionality of the `drop` function.

The `splitAt` function combines these two functionalities by returning both answers in a pair.
That is ; `splitAt n l == ( take n l , drop n l )`

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

§5.3.7. (!!)

The `!!` (read as bang-bang) operator takes a list and a number `n :: Int`, and returns the n^{th} element of the list, counting from `0` onwards.

```
>>> [5,8,3,2,0] !! 0
5
>>> [5,8,3,2,0] !! 1
8
>>> [5,8,3,2,0] !! 2
3
>>> [5,8,3,2,0] !! 3
2
>>> [5,8,3,2,0] !! 4
0
```

But what happens if `n` is not between `0` and `length l`?

Error!

```
>>> [5,8,3,2,0] !! (-1)
*** Exception: Prelude.!!: negative index
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\List.hs:1369:12 in base:GHC.List
  negIndex, called at libraries\base\GHC\List.hs:1373:17 in base:GHC.List
  !!, called at <interactive>:8:13 in interactive:Ghci6

>>> [5,8,3,2,0] !! 5
*** Exception: Prelude.!!: index too large
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\List.hs:1366:14 in base:GHC.List
  tooLarge, called at libraries\base\GHC\List.hs:1376:50 in base:GHC.List
  !!, called at <interactive>:9:13 in interactive:Ghci6
```

So, again, it is advised to avoid using the `!!` operator.

X **Exercise**

Provide a definition for the `!!` operator.

§5.3.8. List `-> Bool`

Functions on lists that return `Bool` are used to check whether lists satisfy certain properties or not. For example -

§5.3.8.1. Elem

The `elem` function is used to determine whether a list contains a particular object.

The `elem` function takes a value and a list, and answers whether the value appears in the list or not, answering in either `True` or `False`.

```
>>> elem 5 [5,8,3,2,0]
True
>>> elem 8 [5,8,3,2,0]
True
>>> elem 3 [5,8,3,2,0]
True
>>> elem 2 [5,8,3,2,0]
True
>>> elem 0 [5,8,3,2,0]
True
```

```
>>> elem 7 [5,8,3,2,0]
False
>>> elem 6 [5,8,3,2,0]
False
>>> elem 4 [5,8,3,2,0]
False
```

And the definition -

λ **elem**

```
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

This reads - " `x` does not appear in the empty list.

`x` appears in a list if and only if it is equal to the first element or it appears somewhere in the rest of the list. "

§5.3.8.2. Generalized Logical Operators

The binary (taking 2 `Bool` inputs) logical operators like `&&` and `||` can be be generalized to take a list of inputs `[Bool]`.

λ **and**

```
and :: [Bool] -> Bool
and (b:bs) = b && ( and bs )
and [] = True
```

The `and` function takes as input a list of type `[Bool]` and answers whether ALL of the elements of the list are `True`.

A few examples -

```
>>> and [True, True, True]
True

>>> and [True, False, True]
False

>>> and []
True

>>> and [False, False, False]
False

>>> and [True && False, True]
False
```

Let's generalize `||` as well.

λ or

```
or :: [Bool] → Bool
or (b:bs) = b || ( or bs )
or [] = False
```

The `or` function takes as input a list of type `[Bool]` and answers whether ANY of the elements of the list is `True`.

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

>>> or [False, False, False]
False

>>> or []
False

>>> or [True, True, False]
True

>>> or [not True, not False]
True
```

X **base cases of list-ary logical operators**

Try to justify why the definitions `and [] = True` and `or [] = False` are required.

§5.4. Strings

A string is how we represent text (like English sentences and words) in programming.

Like many modern programming languages, Haskell defines a string to be just a list of characters.

In fact, the type `String` is just a way to refer to the actual type `[Char]`.

So, if we want write the text "hello there!", we can write it in GHCi as `['h','e','l','l','o',' ','t','h','e','r','e','!']`.

Let's test it out -

```
>>> ['h','e','l','l','o',' ','t','h','e','r','e','!']
"hello there!"
```

But we see GHCi replies with something much simpler - `"hello there!"`

This simplified form is called syntactic sugar. It allows us to read and write strings in a simple form without having to write their actual verbose syntax each time.

So, we can write -

```
>>> "hello there!"
"hello there!"

>>> :type +d "hello there!"
"hello there!" :: String
```

The type `String` is just a way to refer to the actual type `[Char]`.

And since strings are just lists, all the list functions apply to strings as well.

```
>>> 'h' : "ello there!"
"hello there!"

>>> "hello " ++ "there!"
"hello there!"

>>> head "hello there!"
'h'

>>> tail "hello there!"
"ello there!"

>>> take 5 "hello there!"
"hello"

>>> drop 5 "hello there!"
" there!"

>>> elem 'e' "hello there!"
True

>>> elem 'w' "hello there!"
False

>>> "hello there!" !! 7
'h'

>>> "hello there!" !! 6
't'
```

But there are some special functions just for strings -

`words` breaks up a string into a list of the words in it.

```
>>> words "hello there!"
["hello","there!"]
```

And `unwords` combines the words back into a single string.

```
>>> unwords ["hello","there!"]
"hello there!"
```

`lines` breaks up a string into a list of the lines in it.

```
>>> lines "hello there!\nI am coding..."
["hello there!","I am coding..."]
```

Ans `unlines` combines the lines back into a single string.

```
>>> unlines ["hello there!","I am coding..."]
"hello there!\nI am coding...\n"
```

§5.5. Structural Induction for Lists

Suppose we wan prove some fact about lists.

We can use the following version of the ÷ **principle of mathematical induction** -

÷ **structural induction for lists**

Suppose for each list `l` of type `[T]`, we have a statement φ_{l}. If we can pore the following two statements -

- $\varphi_{[]}$
- For each list of the form `(x:xs)`, if φ_{xs} is true, then $\varphi_{(x:xs)}$ is also true.

then φ_{l} for all finite lists `l`.

Let use this principle to prove that

Theorem The definition of `length` terminates on all finite lists.

Proof Let φ_{l} be the statement

The definition of `length l` terminates.

To use ÷ **structural induction for lists**, we need to prove -

- $\langle\langle \varphi_{[]} \rangle\rangle$
  The definition of `length []` directly gives `0`.
- $\langle\langle$ For each list `(x:xs)`, if φ_{xs}, then $\varphi_{(x:xs)}$ also. $\rangle\rangle$
  Assume φ_{xs} is true.
  The definition for `length (x:xs)` is `1 + length xs`.
  By φ_{xs},we know that `length xs` will finally give return some number `n`.
  Therefore `1 + length xs` reduces to `1 + n`.
  And `1 + n` obviously terminates. ∎

§5.6. Sorting

÷ **sorted list**

A list is said to be **sorted**
if and only if
its elements appear in ascending order of their values.

OR EQUIVALENTLY

A list `[x₁, x₂, x₃, ... , xₙ₋₁, xₙ]` is said to be **sorted**
if and only if
$x_1 \leqslant x_2 \leqslant x_3 \leqslant \ldots \leqslant x_{n-1} \leqslant x_n$

OR EQUIVALENTLY

A list $[x_1, x_2, x_3, \ldots, x_{n-1}, x_n]$ is said to be **sorted**
if and only if
$(x_1 \leqslant x_2)$ && $(x_2 \leqslant x_3)$ && $(x_3 \leqslant x_4)$ && ... && $(x_{n-1} \leqslant x_n)$

Here are a few examples of ÷ **sorted lists** -

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

[0, 10, 20, 30, 40]

[-10, -5, 0, 5, 10]

[2, 3, 5, 7, 11, 13, 17]

[100, 200, 300, 400, 500]

[-100, -50, -10, -1, 0, 1, 10]

[1, 1, 2, 3, 5, 8, 13]
```

and here are few which are NOT ÷ **sorted lists** -

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

[30, 10, 40, 0, 20]

[10, -5, 0, -10, 5]

[11, 2, 17, 5, 13, 3, 7]

[500, 100, 300, 200, 400]

[10, -1, -100, 0, -50, -10, 1]

[8, 1, 13, 5, 3, 1, 2]
```

Let's write a function that takes a list of and answers whether it is a ≑ **sorted list** or not -

```
isSorted [] = True
isSorted [x] = True
isSorted (x:x':xs) = ( x ⩽ x' ) && ( isSorted (x':xs) )
```

This reads - " A list which contains nothing is a ≑ **sorted list**.
A list containing exactly one element is also a ≑ **sorted list**.
If the first two elements of the list are `x` and `x'` and the rest of the elements form a list `xs`, then the list is sorted if and only if
`x ⩽ x'` AND the list `x':xs` (i.e., the λ **tail of list** `x:x':xs`, the given input) is sorted. "

Now we introduce an infamous problem in computer science, "sorting"!

≑ **sorting**

Sorting is the act of taking a given list and rearranging the contained elements so that it becomes a ≑ **sorted list**.

In Haskell, we do this using the `sort` function.

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

>>> sort [30, 10, 40, 0, 20]
[0,10,20,30,40]

>>> sort [10, -5, 0, -10, 5]
[-10,-5,0,5,10]

>>> sort [11, 2, 17, 5, 13, 3, 7]
[2,3,5,7,11,13,17]

>>> sort [500, 100, 300, 200, 400]
[100,200,300,400,500]

>>> sort [10, -1, -100, 0, -50, -10, 1]
[-100,-50,-10,-1,0,1,10]

>>> sort [8, 1, 13, 5, 3, 1, 2]
[1,1,2,3,5,8,13]
```

Let us see whether we can define the `sort` function.

Well, it is obvious that

```
sort [] = []
```

So we are left with defining `sort (x:xs)`.

In the style of recursive definitions, we can assume that we already have `sort xs` computed.

i.e., let us define

```
sortedTail = sort xs
```

and we can henceforth use `sortedTail` to refer to `sort xs`.

Now, `sortedTail`, being `sort xs`, contains all the elements of `xs`, rearranged in ascending order. But it doesn't contain `x`.

If we were able to include `x` in `sortedTail` without disturbing this ascending order, we would be done!

So let's do that -

First we take those elements of `sortedTail` which should appear **before** `x` in the ascending order. (i.e., the elements `< x`)

```
[e | e ← sortedTail , e < x]
```

Then we follow that with `x` itself.

```
[e | e ← sortedTail , e < x] ++ [x]
```

And then we add the elements of `sortedTail` that should appear **after** `x` in the ascending order. (i.e., the elements `≥ x`)

```
[e | e ← sortedTail , e < x] ++ [x] ++ [e | e ← sortedTail , e ≥ x]
```

And thus we obtain a list containing `x` as well as all the elements of `sortedTail`, arranged in ascending order, i.e., `sort (x:xs)`

Putting it all together, we can write a definition for `sort` as follows -

λ sort

```
sort []     = []
sort (x:xs) = let sortedTail = sort xs in
 [e | e ← sortedTail , e < x] ++ [x] ++ [e | e ← sortedTail , e ≥ x]
```

Let's see an example computation -

```
  sort [5, 1, 13, 8, 3, 1, 2]

= let sortedTail = sort [1, 13, 8, 3, 1, 2] in

   [e | e ← sortedTail , e < 5] ++ [5] ++ [e | e ← sortedTail , e ⩾ 5]

= let sortedTail = [1,1,2,3,8,13] in

   [e | e ← sortedTail , e < 5] ++ [5] ++ [e | e ← sortedTail , e ⩾ 5]

= let sortedTail = [1,1,2,3,8,13] in

   [1,1,2,3] ++ [5] ++ [e | e ← sortedTail , e ⩾ 5]

= let sortedTail = [1,1,2,3,8,13] in

   [1,1,2,3] ++ [5] ++ [8,13]

= [1,1,2,3] ++ [5] ++ [8,13]

= [1,1,2,3,5,8,13]
```

§5.7. Optimization

Suppose we want to reverse the the order of elements in a list.

For example, transforming the list `[5,8,3,2,0]` into `[0,2,3,8,5]`.

So how do we define the function `reverse`?

An obvious definition is -

λ naive reverse

```
reverse []     = []
reverse (x:xs) = ( reverse xs ) ++ [x]
```

But this is not "optimal".

What does this mean? Let's see -

Let's apply the definitions of `reverse` and `(++)` to see how `reverse [5,8,3]` is computed -

```
reverse [5,8,3] == ( reverse [8,3]  ) ++ [5]

                == ( ( reverse [3]  ) ++ [8] ) ++ [5]

                == ( ( ( reverse [] ) ++ [3] ) ++ [8] ) ++ [5]

                == ( ( [] ++ [3] ) ++         [8]    ) ++         [5]

                == (         [3]    ++         [8]    ) ++         [5]

                == (          3     :  ( [] ++ [8] ) ) ++         [5]

                == (          3     :          [8]    ) ++         [5]

                == (          3     : (        [8]      ++         [5] )

                ==            3     : (         8       :  ( [] ++ [5] ) )

                ==            3     : (         8       :          [5]   )

                -- which finally is
                        [3,8,5]
```

So we see that this takes 10 steps of computation.

Let us take an alternative definition of `reverse` -

optimized reverse

```
reverse l = help [] l where
    help xs (y:ys) = help (y:xs) ys
    help xs []     = xs
```

Let us how this one is computed step by step -

```
reverse [5,8,3] == help      [] [5,8,3]

                == help     [5] [8,3]

                == help   [8,5] [3]

                == help [3,8,5] []

                == [3,8,5]
```

So we see this computation takes only 5 steps, as compared to 10 from last time.

So, in some way, the second definition is better as it requires much less steps.

We can comment on something similar for `splitAt`

naive splitAt

```
splitAt n l = ( take n l , drop n l )
```

optimized splitAt

```
splitAt n []     = ( []   , [] )
splitAt n (x:xs) = ( x:ys , zs ) where
    (ys,zs) = splitAt (n-1) xs
```

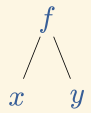
Exercise

(1) Prove that the two definitions are equivalent using **structural induction for lists**.

(2) See which definition takes more steps to compute `splitAt 2 [5,8,3]`

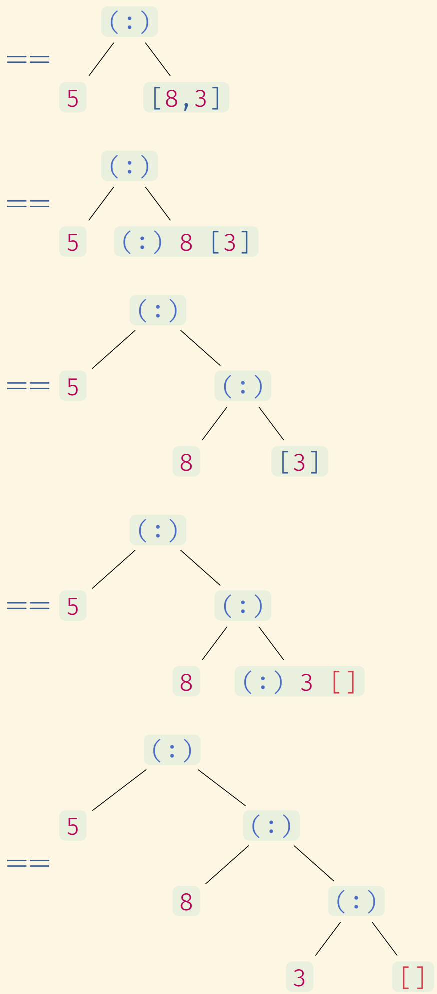
§5.8. Lists as Syntax Trees

Recall **abstract syntax tree**.

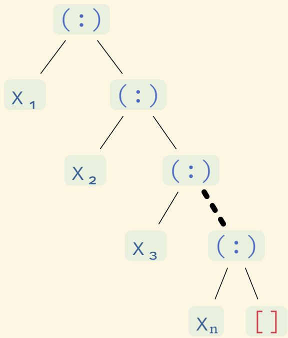
Remember that we represent $f(x, y)$ as 

Using this rule, see whether the following steps make sense -

`[5,8,3]` == `(:) 5 [8,3]`



In fact any list `[x₁,x₂,x₃, … ,xₙ]` can be represented as



This is the representation that Haskell actually uses to store lists.

§5.9. Dark Magic

We can use our arithmetic progression notation to generate infinite arithmetic progressions.

```
>>> [0..]
[0,1,2,3,4,5,6,7,8,9, ... ]

>>> [2,5..]
[2,5,8,11,14,17,20,23,26,29, ... ]
```

We can define infinite lists like -

a list of infinitely many `0`s -

```
zeroes = 0 : zeroes
```

```
>>> zeroes
[0,0,0,0,0,0,0,0,0,0, ... ]
```

the list of all natural numbers -

```
naturals = l 0 where l n = n : l (n+1)
```

```
>>> naturals
[0,1,2,3,4,5,6,7,8,9, ... ]
```

and the list of all fibonacci numbers -

```
fibs = l 0 1 where l a b = a : l b (a+b)
```

```
>>> fibs
[0,1,1,2,3,5,8,13,21,34, ... ]
```

Since we obviously cannot view the entirety of an infinite list, it is advisable to use the λ **take from list** function `take` to view an initial section of the list, rather than the whole thing.

§5.9.1. Exercises

Balloons

In an ICPC contest, balloons are distributed as follows:
- Whenever a team solves a problem, that team gets a balloon.
- The first team to solve a problem gets an additional balloon.

A contest has 26 problems, labelled $A, B, ..., Z$. You are given the order of solved problems in the contest, denoted as a string s, where the i-th character indicates that the problem s_i has been solved by some team. No team will solve the same problem twice.

Write a function `balloons :: String → Int` to determine the total number of balloons used in the contest. Note that some problems may be solved by none of the teams.

Example :

```
balloons "ABA" = 5
balloons "A" = 2
balloons "ORZ" = 6
balloons "BAAAA" = 7
balloons "BAAAA" = 7
balloons "BKPT" = 8
balloons "BKPT" = 8
balloons "HASKELL" = 13
```

Neq Array (INOI 2025 P1)

Given a list A of length N, we call a list of integers B of length N such that:
- All elements of B are positive, ie $\forall 1 \leq i \leq N, B_i > 0$
- B is non-decreasing, ie $B_1 \leq B_2 \leq ... \leq B_N$
- $\forall 1 \leq i \leq N, B_i \neq A_i$

Let $\text{neq}(A)$ denote the minimum possible value of the last element of B for a valid array B.

Write a function `neq :: [Int] → Int` that takes a list A and returns the $\text{neq}(A)$.

Example :

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

X Kratki (COCI 2014)

Given two integers N and K, write a function `krat :: Int → Int → Maybe [Int]` which constructs a permutation of numbers from 1 to N such that the length of its longest monotone subsequence (either ascending or descending) is exactly K or declare that the following is not possible.

A monotone subsequence is a subsequence where elements are either in non-decreasing order (ascending) or non-increasing order (descending).

Example:

```
krat 4 3 = Just [1,4,2,3]
krat 5 1 = Nothing
krat 5 5 = Just [1,2,3,4,5]
```

For example 1: The permutation (1, 4, 2, 3) has longest ascending subsequence (1, 2, 3) of length 3, and no longer monotone subsequence exists. For example 2: It's impossible to create a permutation of 5 distinct numbers with longest monotone subsequence of length 1. For example 3: The permutation (1, 2, 3, 4, 5) itself is the longest monotone subsequence of length 5.

X Putnik (COCI 2013

Chances are that you have probably already heard of the traveling salesman problem. If you have, then you are aware that it is an NP-hard problem because it lacks an efficient solution. Well, this task is an uncommon version of the famous problem! Its uncommonness derives from the fact that this version is, actually, solvable.

Our vacationing mathematician is on a mission to visit N cities, each exactly once. The cities are represented by numbers $1, 2, \ldots, N$. What we know is the direct flight duration between each pair of cities. The mathematician, being the efficient woman that she is, wants to modify the city visiting sequence so that the total flight duration is the minimum possible.

Alas, all is not so simple. In addition, the mathematician has a peculiar condition regarding the sequence. For each city labeled K must apply: either all cities with labels smaller than K have been visited before the city labeled K or they will all be visited after the city labeled K. In other words, the situation when one of such cities is visited before, and the other after is not allowed.

Assist the vacationing mathematician in her ambitious mission and write a function `time :: [[Int]] → Int` to calculate the minimum total flight duration needed in order to travel to all the cities, starting from whichever and ending in whichever city, visiting every city exactly once, so that her peculiar request is fulfilled. Example :

```
time [
  [0,5,2],
  [5,0,4],
  [2,4,0]] = 7


time [
  [0,15,7,8],
  [15,0,16,9],
  [7,16,0,12],
  [8,9,12,0]] = 31
]
```

In the first example: the optimal sequence is 2, 1, 3 or 3, 1, 2. The sequence 1, 3, 2 is even more favourable, but it does not fulfill the condition. In the second example: the sequence is either 3, 1, 2, 4 or 4, 2, 1, 3.

X Look and Say

Look-and-say sequences are generated iteratively, using the previous value as input for the next step. For each step, take the previous value, and replace each run of digits (like 111) with the number of digits (3) followed by the digit itself (1).

For example:

- 1 becomes 11 (1 copy of digit 1).
- 11 becomes 21 (2 copies of digit 1).
- 21 becomes 1211 (one 2 followed by one 1).
- 1211 becomes 111221 (one 1, one 2, and two 1s).
- 111221 becomes 312211 (three 1s, two 2s, and one 1).

Write function `lookNsay :: Int → Int` which takes an number and generates the next number in its look and say sequence.

X Triangles (Codeforces 1119E)

Pavel has several sticks with lengths equal to powers of two. He has a_0 sticks of length $2^0 = 1$, a_1 sticks of length $2^1 = 2$, ..., a_n sticks of length 2^n.

Pavel wants to make the maximum possible number of triangles using these sticks. The triangles should have strictly positive area, each stick can be used in at most one triangle.

It is forbidden to break sticks, and each triangle should consist of exactly three sticks. Write a function `triangles :: [Int] → Int]` to find the maximum possible number of triangles.

Examples

```
triangles [1,2,2,2,2] = 3
triangles [1,1,1] = 0
triangles [3,3,3] = 1
```

In the first example, Pavel can, for example, make this set of triangles (the lengths of the sides of the triangles are listed): $(2^0, 2^4, 2^4)$, $(2^1, 2^3, 2^3)$, $(2^1, 2^2, 2\hat{}2)$.

In the second example, Pavel cannot make a single triangle.

In the third example, Pavel can, for example, create this set of triangles (the lengths of the sides of the triangles are listed): $(2^0, 2^0, 2^0)$, $(2^1, 2^1, 2^1)$, $(2^2, 2^2, 2^2)$.

X Thanos Sort (Codeforces 1145A)

Thanos sort is a supervillain sorting algorithm, which works as follows: if the array is not sorted, snap your fingers* to remove the first or the second half of the items, and repeat the process.

Given an input list, what is the size of the longest sorted list you can obtain from it using Thanos sort? Write function `thanos :: Ord a ⇒ [a] → Int` to determine that.

* Infinity Gauntlet required.

Examples

```
thanos [1,2,2,4] = 4
thanos [11, 12, 1, 2, 13, 14, 3, 4] = 2
thanos [7,6,5,4] = 1
```

In the first example the list is already sorted, so no finger snaps are required.

In the second example the list actually has a sub-array of 4 sorted elements, but you can not remove elements from different sides of the list in one finger snap. Each time you have to remove either the whole first half or the whole second half, so you'll have to snap your fingers twice to get to a 2-element sorted list.

In the third example the list is sorted in decreasing order, so you can only save one element from the ultimate destruction.

Deadfish

Deadfish XKCD is a fun, unusual programming language. It only has one variable, called s, which starts at 0. You change s by using simple commands.

Your task is to write a Haskell program that reads Deadfish XKCD code from a file, runs it, and prints the output.

Deadfish XKCD has the following commands:

Command	What It Does
x	Add 1 to s.
k	print s as a number.
c	Square s,
d	Subtract 1 from s.
X	Start defining a function (the next character is the function name).
K	Print s as an ASCII character.
C	End the function definition or run a function.
D	Reset s back to 0.
{}	Everything inside curly braces is considered a comment.

Extra Rules:

- s must stay between 0 and 255.
- If s goes above 255 or below 0, reset it back to 0.
- Ignore spaces, newlines, and tabs in the code.
- Other characters (not commands) work differently depending on the subtask.

While you can do the whole exercise in one go, we recommend doing the following subtasks in order.

1. Basic (x, k, c, d only)

```
run "xxcxkdk" = "54"
```

2. Extended (add K, D)

```
run "xxcxxxxcxxxxxxxxK" = "H"
```

3. Functions (all commands)

```
run "XUxkCxxCUCUCU" = "345"
```

4. Comments (ignore {})

Ignore content inside curly braces.

Hint: Don't be afraid to use tuples!

X Weakness and Poorness (Codeforces 578C)

You are given a sequence of n integers $a_1, a_2, ..., a_n$.

Write a function `solve :: [Int] → Float` to determine a real number x such that the weakness of the sequence $a_1 - x, a_2 - x, ..., a_n - x$ is as small as possible.

The weakness of a sequence is defined as the maximum value of the poorness over all segments (contiguous subsequences) of a sequence.

The poorness of a segment is defined as the absolute value of sum of the elements of segment.

Examples

```
solve [1,2,3] = 2.0
solve [1,2,3.4] = 2.5
```

Note For the first case, the optimal value of x is 2 so the sequence becomes $-1, 0, 1$ and the max poorness occurs at the segment -1 or segment 1.

For the second sample the optimal value of x is 2.5 so the sequence becomes $-1.5, -0.5, 0.5, 1.5$ and the max poorness occurs on segment $-1.5, -0.5$ or $0.5, 1.5$.

X Group

Write function `group :: [a] → [[a]]` which groups adjacent equal elements. For example:

```
group [1,1,1,2,4,8,8,8,8,10,10] = [[1,1,1],[2],[4],[8,8,8,8],[10,10]]
group "haskell" = ["h","a","s","k","e","ll"]
```

Polymorphism and Higher Order Functions

§6.1. Polymorphism

§6.1.1. Classification has always been about *shape* and *behvaiour* anyway

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 property of elements, that is, how they look like, and the functions defined on them.

And we have seen how to define functions from a given type to another given type using the above ideas, 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 *shape* to the elements of a type, in the following sense:

Consider the type `[Integer]`, 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.

- A function for lists would thus have 2 components, at least conceptually if not explicit in the code itself:
  - The first being that of a list, which can be interacted with using functions like `head`.
  - The second being that of `Integer`, So that functions on `Integer` can be applied to the elements of the list.

  consider the following example:

  λ squaring all elements of a list
  ```
  squareAll :: [Integer] → [Integer]
  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
  ```
  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:

- `elem`, which checks in an element belong to a list.
- `(==)`, which checks if 2 elements are equal.
- `drop`, which takes a list and discards a specfied about of items in the list from the beginning.

These functions seem to note care about all of the properties (shape and behaviour together) of their inputs.

- The `elem` function wants its inputs to be list 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, this is the only behaviour it is interested in. (A counter example would be the type of functions: `Integer → Integer`, and we will discuss why this is the case soon.)
- The `drop` function just cares about the list structre of an element, and does not look at the behaviour of the list items at all.

To define any function in haskell, one needs to give them a type, haskell demands so, so lets look at the case of the `drop` function. One possible way to have it would be to define one for every single type, as shown below:

```
dropIntegers :: Integer → [Integer] → [Integer]
dropIntegers = ...
dropChars :: Integer → [Char] → [Char]
dropChars = ...
dropBools :: Integer → [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.
    - Differential Equations let us talk about "change" in many different scenarios.

    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 different input types (not necessarily all types, restrictions are allowed).
- Its output type can be different for different inputs types.

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

λ drop
```
drop :: Integer → [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 :: Integer → [a] → [a]` where we have used the variable `a` (usually called a type variable) instead of explicitly mentioning a type.

The goal of polymorphic functions is to let us **abstract** over a collection of types. That take a collection of types, based on some common property (either shape, or behaviour, maybe both) and treat that as a collection of elements. This lets us build functions that work on "all lists" or "all maybe types" and so on.

The example λ **drop** brings together all types of lists and only looks at the *shape* of the element, that of a list, and does not look at the bhevaiour at all. This is shown by using the type variable `a` in the definition, indicating that we don't care about the properties of the list items.

X 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`.

We have now given a type to one of the 3 functions discussed above, by giving a way to group together types by their common *shape*. This is not enough to give types of the other two functions (`(==)` and `elem`), to do so we define the following:

≑ Behaviour

Given a type `T`, the **behaviour** of the elements in `T` is the set of definable functions whose type includes `T`.

We use this to define the two types of polymorphism, one of which we have already seen in this section, and we will look at the other one more deepy in the next.

≑ 2 Types of Polymorphism

- Polymorphism done by grouping types that with common ***shape*** is called **Parametric Polymorphism**.
- Polymorphism done by grouping types that with common ***behaviour*** is called **Ad-Hoc Polymorphism**.

We will come back to **parametric polymorphism** in the second half of the chapter, but for now we discuss **Ad-Hoc polymorphism**.

§6.1.2. A Taste of Type Classes

Consider the case of the `Integer` functions

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

g :: Integer → Integer
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, so mathematicans call them the same, and write $f = g$ 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, we care about the way things are computed, 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 would want to factor into what they mean by "sameness". So maybe the assumption that 2 functions being equal pointwise imply the functions are equal is not 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 (Integer → Integer)' arising from a use of '=='
      ... more error
  |
8 | h = f == g
  |
```

To tackle the problem of giving a type for `(==)`, we define the following:

÷ **Typeclasses**

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

The previous section talked about grouping types together by the common *shape* of the elements but 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
elem _ []       = False
elem e (x : xs) = e == x || elem e xs
```

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.

X Shape is behaviour?

The two types of polymorphism, that is parametric and ad-hoc, are not exlusive, there are plenty of function where both are seen together, an example would be `elem`.

These two happen to not be that different conceptually either, we give elements their *shape* using functions, try figuring out what the functions are for list types, maybe type, tuples and either type.

That being said, the syntax used to define parametric polymorphism sets us to set operations while defining the type of the function which is very powerful.

§6.2. Higher Order Functions

One of the most powerful features of functional programming languages is that it lets one pass in functions as argument to another function, and have functions 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.

§6.2.1. Currying

Perhaps the first place where we have encountered higher order functions is when we defined `(+) :: Int → Int → Int` way back in Section §3.4.. 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 curry $: C^{A \times B} \to \left(C^B\right)^A$, and then defining its inverse.

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

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

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

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

X **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, that is, 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:
- 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 different 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:

λ curry and uncurry
```
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

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

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

We will find many more examples in the next section.

§6.2.2. 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
```
(.) :: (b → c) → (a → b) → (a → c)
g . f = \a → g (f a)

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

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

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

Breaking a complicated function into simpler parts, and being able to combime them is fairly 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 `Integer → Integer` and `Integer → Bool`.

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

λ function application function
```
($) :: (a → b) → a → b
f $ a = f a

(&) :: a → (a → b) → b
a & f = f a
```

These may seem like a fairly trivial function that really doesn't offer anything apart from an extra `$`, but the following 3 lines make them useful

λ operator precedence
```
-- The 'r' in infixr says a.b.c.d is interpreted by haskell as a.(b.(c.d))
infixr 9 .
infixr 0 $
infixl 1 &
```

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

-- also
x & f & g & h & i
```

which in my opinion is much simpler to read!

Exercise

Write a function `apply_n_times` 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: `apply_n_times (+1) 5 3` would return `8`. Also figure out the type of the function.

§6.2.3. 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 do so for an analysis that happens before running 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.

§6.2.4. Higher Order Functions on Maybe Type : A Case Study

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

As a refresher on **Maybe Types**, given a type `a`, one can add an *extra element* to it by making it the type `Maybe a`. For example, given the type `Integer`, whose elements are all the integers, the type `Maybe Integer` will be the collection of integers along with an extra element, which we call `Nothing`.

Maybe Types are meant to capture failure, for example, the λ **function to a maybe type** defines the `reciprocal` function, which takes a rational number, and returns its reciprocal, except when the input is `0`, in which case it returns the *extra value* which is `Nothing`.

To state that elements belong to a **Maybe Type** they are decorated with `Just`. For example:

- The type of `5` is `Integer`
- The type of `Just 5` is `Maybe Integer`.

To see an example of some functions that use `Maybe` in their type definitions are:

- A safe version of `head` and `tail`:
  - `safeHead :: [a] -> Maybe a`
  - `safeTail :: [a] -> Maybe [a]`
- A safe way to index a list, that is a safe version of `(!!)`:
  - `safeIndex :: [a] -> Int -> Maybe a`

X Safety First

Define the functions `safeHead`, `safeTail` and `safeIndex`.

Something that should be noted is that so far in the book, `head`, `tail` and `(!!)` are the only functions for which we need safe versions. This is because these are the only functions that are not defined for all possible inputs and can hence give an error while the program executes (that would be like passing empty list to head, or idexing an element at a negative position). Every other function we have seen will always have a valid output, that is, it is literally impossible for functions to fail for not having a valid input if one only uses safe functions!

This may seem like a fairly trivial fact for those who are learning haskell as thier first programming language, but for those who has programmed in languages like Java, Python, C or so on, it is impossible to write a program that would lead to an error which is equivalent to the following:

- Nonetype does not have this attribute: Python
- Null Pointer Exception: Java
- Memory Access Violation or Segfault for derefencing a null pointer: C

If these erros have haunted you, you have our condolences, all of these would have been completely avoided if the langauge had some version of `Maybe`, or even some bare bones type system in case of python.

All of the safety provided by `Maybe` types has 1 potential drawback: When using `Maybe` types, one eventually runs into a problem that looks something like this:

- While solving a complicated problem, one would break it down into simpler parts, that would correspond to many tiny functions, that will come to gether to form the functions which solves the problem.
- Turns out that one the functions, maybe something in the very beginning returns a `Maybe Integer` instead of an `Integer`.
- This means that the next function along the chain, would have had to have its input type as `Maybe Integer` 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**

```haskell
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f (Just a) = Just . f $ a
maybeMap _ Nothing  = Nothing

(<$>) :: (a -> b) -> Maybe a -> Maybe b -- symbol version
f <$> 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:

```haskell
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!

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

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.

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:

```
f :: a -> Maybe b
g :: b -> Maybe c

h x = g <$> f x :: a -> Maybe (Maybe c)
```

This is most likely undesirable, the point of `Maybe` was to say that there is a possiblility of error, the point of `(<$>)` was to propogate that possible error then the type `Maybe (Maybe c)` seems to not have a place here.

To rectify this, we find a way to compose such functions together:

```
maybe_comp :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
infixr 8 >=>
```

This cute looking function is called the **fish** operator. This will be our way to compose functions of the shape `a -> Maybe b` together, but note that the order of inputs is reversed, so it not looks like a pipe through which the value is passed. The above function `h` is defined as follows:

```
h = f >=> g :: a -> Maybe c
```

This function, takes a value of type `a`, first applies `f` to it, and then applies `g` to it in a way that the final output is of type `Maybe c`, and of course, we can use this to make longer chains!

```
func1 :: a -> Maybe b
func2 :: b -> Maybe c
func3 :: c -> Maybe d
func4 :: d -> Maybe e

final :: a -> Maybe e
final = func1 >=> func2
    >=> func3 >=> func4
```

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

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

Contents

§6.3. Exercise

Guard Idiom

(i) Sometimes we have a boolean check that decides whether the return value is a failure or success. Write a function `ensure :: Bool → a → Maybe a` which returns `Nothing` if the boolean is `False` and `Just inp` when the boolean is `True` and `inp` is the other input.

(ii) Write a function `guard :: Bool → Maybe ()` which gives `Nothing` when the boolean is `False` and `Just ()` when it is `True`

(iii) Write an operator `($>) :: Maybe a → b → Maybe b` which is a no-op on `Nothing` values, but replaces whatever is inside a `Just` value on the left with the value on the right.

(iv) Can you now write `ensure` using only `guard` and `$>`? This is called the Guard-Sequence idiom and is extremely common in production level Haskell code.

(v) While we don't use it here, could you define `(*>) :: Maybe a → Maybe b → Maybe b` which returns `Nothing` and returns the second argument if the first argument is `Nothing` or a `Just` value respectively. This is also used in tandem with `guard`.

[9]

Some List Functions

(i) Define `filter :: (a → Bool) → [a] → [a]` which given a predicate and list of elements, returns the list of elements satisfying the predicate.

(ii) Define `map :: (a → b) → [a] → [b]` which given a function and a list of elements, applies the function to each element and returns the new list.

(iii) Define `concatMap :: (a → [b]) → [a] → [b]` which maps a function over all the elements of a list and concatenate the resulting lists. Do not use `map` in your definition.

(iv) Define `groupBy :: (a → a → Bool) → [a] → [[a]]` which groups adjacent elements according to some relation. In last chapter, we have seen `group` which is nothing but `groupBy (==)`. We could also have `groupBy (≤)` to get the consecutive increasing subsequences.

[9]The `$>` and `*>` are part of the `Data.Functors` module and `guard` is part of the `Control.Monad` module. Their actual type signatures work for any functor, not just `Maybe`. We will see what functors are in later chapters.

X Conditional Apply

(i) Write a function `applyWhen :: Bool → (a→ a) → a → a` which applies a function to a value if a condition is true, otherwise, it returns the value unchanged.

(ii) Define `on :: (b → b → c) → (a → b) → a → a → c` such that `on b u x y` runs the binary function `b` on the results of applying unary function `u` to two arguments `x` and `y`. This is again quite common in production level code as it avoids rewriting the same function over and over.

(iii) Prove that

```
applyWhen True = id
applyWhen False f = id
```

(iv) Prove that

```
(*) `on` id = (*)
((*) `on` f) `on` g = (*) `on` (f . g)
flip `on` f . flip `on` g = flip `on` (g . f)
```

X Theorems for Free

We will talk about some of the theorems in Wadler's iconic paper "Theorems for Free". From the type of a polymorphic function, we can derive a theorem which all such functions will follow.

(i) Given `f :: a1 → b1` and `g :: a2 → b2`, prove that `const (f a1) (g a2) = f (const a1 a2)`

(ii) Given `r :: [a] → [a]` and `f :: b → c`, prove that `map f . r = r . map f`

(iii) Define `prodMap :: (a → a1) → (b → b1) → (a,b) → (a1, b1)` and `coProdMap :: (a → a1) → (b → b1) → Either a b → Either a1 b1` which apply two given functions to the elements of a tuple or an `Either`.

(iv) Given `r :: (a,b) → (a,b)` and `f :: a → a1, g :: b → b1`, prove that `r . prodMap f g = prodMap f g . r`

(v) Given `r :: Either a b → Either a b` and `f :: a → a1, g :: b → b1`, prove that `r . coProdMap f g = coProdMap f g . r`

Can you guess the general scheme for the theorem we can get for free? Could you prove your hypothesis?

X Product and Co-Products

(i) Define the function `product :: (a → b) → (a → c) → a → (b, c)` which takes two functions $f : x \mapsto fx$ and $g : x \mapsto gx$ and returns a function $f \times g : x \mapsto (fx, gx)$.

(ii) Define a function `coProduct :: (b → a) → (c → a) → Either b c → a` which takes two functions f', g' from different domains but same co-domain and combines them.

One can make a commutative diagram for these functions as follows:

$$
\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dd, "g"'] \arrow[dr, dashed, "\texttt{product f g}"] & & B \\
& B \times C \arrow[ur, "\texttt{fst}"'] \arrow[dl, "\texttt{snd}"] & \\
C & &
\end{tikzcd}
$$

$$
\begin{tikzcd}
A & & B \arrow[ll, "f^{-1}"] \arrow[dl, "\texttt{Left}"] \\
& \texttt{Either B C} \arrow[ul, dashed, "\texttt{coProduct f' g'}"'] & \\
C \arrow[uu, "g^{-1}"] \arrow[ur, "\texttt{Right}"] & &
\end{tikzcd}
$$

Considering the 'co' prefix is used to define a talk about the dual of a function, could you guess what a dual means? A hint could be the fact that `Either` can be called a co-tuple as well as f^{-1} can be called co-f.

X Composing Compose

(i) Infer the type of `(.).(.)` manually. Can you see the use case? This is often defined in production level Haskell as `.: = (.).(.)` and is called the Blackbird Combinator (It also has another name but as it is much more explicit, we leave it upto your curiosity).

(ii) Can you guess the type of `(.).(.).(.)`? Now by induction, what is the type of a similar expression with n many `(.)`?