

Haskell for CMI

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

© 2025 Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

Text licensed under CC-by-SA-4.0

Code licensed under AGPL-3.0

This is (still!) an incomplete draft.

Please send any corrections, comments etc. to feedback_host@mailthing.com

Last updated July 06, 2025.

To someone

Basic Theory

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

§1.1. Precise Communication (better name suggestion always welcome)

Haskell (along a lot of programming) 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 context of Haskell, computers, being machines, are extremely stupid. Unless we spell out the details for them in excruciating detail, they are not going to understand what we want them to do.

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

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

§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
 - $x^3 \cdot x^5 + x^2 + 1$
 - $f(g(x, y), f(a, h(v), c), h(h(h(n))))$
 - $\forall \alpha \in \mathbb{R} \exists L \in \mathbb{R} \forall \varepsilon > 0 \exists \delta > 0 \mid x - \alpha \mid < \delta \Rightarrow \mid f(x) - f(\alpha) \mid < \varepsilon$
 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 : $\mathbb{R} \rightarrow \mathbb{R}$
- The number of non-trivial zeroes of the Riemann Zeta function

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

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

§1.4. Variables

≡ mathematical variable

A mathematical **variable** is a symbol or chain of symbols meant to represent an arbitrary

≡ mathematical value,

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.

The following example should clarify further.

For example, consider the following theorem -

Theorem Adding 1 to a natural number makes it bigger.

Proof Take n to be an arbitrary natural number.

We know that $1 > 0$.

Adding n to both sides of the preceding inequality yields

$$n + 1 > n$$

Hence Proved !! ■

Here, n is a **mathematical variable** as it isn't any one specific value, but rather **represents an arbitrary** natural number.

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

§1.5. Well-Formed Expressions

Consider the expression -

$$xyx \Leftarrow \forall \Rightarrow f(\curvearrowright \vec{v}$$

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.

⚡ well-formed mathematical expression

Well-formed expressions, like *love*, is one of those things which is easier to identify than to describe.

The following is a procedure to check if a given expression e is **well-formed**:

- first check whether e is a:
 - ⚡ **mathematical value**, or;
 - ⚡ **mathematical variable**
 in which cases e passes the check and is an expression, otherwise;
- check whether e is of the form $f(e_1, e_2, e_3, \dots, e_n)$, where
 - f is a function (which can be either a ⚡ **mathematical value** or ⚡ **mathematical variable**)
 - which takes n inputs,
 - and
 - $e_1, e_2, e_3, \dots, e_n$ are all **well-formed expressions** which are **valid inputs** to f .

Now we can define a **Well-formed expressions** as any expression that satisfies our procedure.

Remark: (the function f can be a ⚡ **mathematical value** or ⚡ **mathematical variable**)

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

$x^3 \cdot x^5 + x^2 + 1$ is $+$ applied to the inputs $x^3 \cdot x^5$ and $x^2 + 1$.

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

$x^3 \cdot x^5$ is \cdot applied to the inputs x^3 and x^5 .

Thus we need to check that x^3 and x^5 are well-formed expressions.

x^3 is $()^3$ applied to the input x .

Thus we need to check that x is a well-formed expression.

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

x^5 is $()^5$ applied to the input x .

Thus we need to check that x is a well-formed expression.

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

$x^2 + 1$ is $+$ applied to the inputs x^2 and 1 .

Thus we need to check that x^2 and 1 are well-formed expressions.

x^2 is $()^2$ applied to the input x .

Thus we need to check that x is a well-formed expression.

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

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

Done!

x checking well-formedness of mathematical expression

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

§1.6. Defining Functions

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

Thus, it is very helpful to have a deeper understanding of how they are defined.

§1.6.1. Using Expressions


In its simplest form, a definition of a function is made up of a left-hand side, ‘ $:=$ ’ in the middle¹, and a right-hand side.

A few examples -

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

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.

A few examples -

- $f(x) := x^3 \cdot x^5 + x^2 + 1$
- $\text{snd}(a, b) := b$
- $\zeta(s) := \sum_{n=1}^{\infty} \frac{1}{n^s}$

§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 -

$$\text{sine} : \mathbb{R} \rightarrow \mathbb{R}$$

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

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

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 this using “where” instead of “let”.

¹In 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 ”.

$$\text{sine} : \mathbb{R} \rightarrow \mathbb{R}$$

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

,where

$T :=$ a right-angled triangle with one angle $= \theta$

$p :=$ the length of the perpendicular of T

$h :=$ the length of the hypotenuse of T

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”.

§1.6.2.2. Anonymous Functions

A function definition such as

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

$$f(x) := x^3 \cdot x^5 + x^2 + 1$$

for convenience, can be rewritten as -

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

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 $:=$

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 -

$$\text{harmonicSum} : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$$

$$\text{harmonicSum}(x, y) := \frac{1}{x} + \frac{1}{y}$$

which, for convenience, can be rewritten as -

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

§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 -

$$\langle \text{functionName} \rangle (x) := \begin{cases} \langle \text{expression}_1 \rangle ; \text{if } \langle \text{condition}_1 \rangle > \\ \langle \text{expression}_2 \rangle ; \text{if } \langle \text{condition}_2 \rangle > \\ \langle \text{expression}_3 \rangle ; \text{if } \langle \text{condition}_3 \rangle > \\ \vdots \\ \langle \text{expression}_n \rangle ; \text{if } \langle \text{condition}_n \rangle > \end{cases}$$

For example, consider the following definition -

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

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 -

$$\begin{aligned} \text{signum}(0) &:= 0 \\ \text{signum}(x) &:= \frac{x}{|x|} \end{aligned}$$

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 -

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

§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, $f(4)$.

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**

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

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

then all the statements $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \dots$ 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.

✖ **Exercise**

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

§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 -

- $\langle\langle \varphi_0 \rangle\rangle$
The only natural number which is ≤ 0 is 0, and $F(0) := 1$, so the definition terminates immediately.
- $\langle\langle \text{For each } n > 0, \text{ if } \varphi_{n-1} \text{ is true, then } \varphi_n \text{ is also true.} \rangle\rangle$
Assume that φ_{n-1} is true.
Let m be an arbitrary natural number which is $\leq n$.
 - $\langle\langle \text{Case 1 } (m \leq 1) \rangle\rangle$
 $F(m) := 1$, so the definition terminates immediately.
 - $\langle\langle \text{Case 2 } (m > 1) \rangle\rangle$
 $F(m) := F(m-1) + F(m-2)$,
and since $m-1$ and $m-2$ are both $\leq n-1$,
 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+1} is proved!

Hence the theorem is proved!! ■

§1.7. Trees

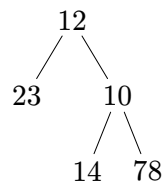
Trees are a way to structure a collection of objects.

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.

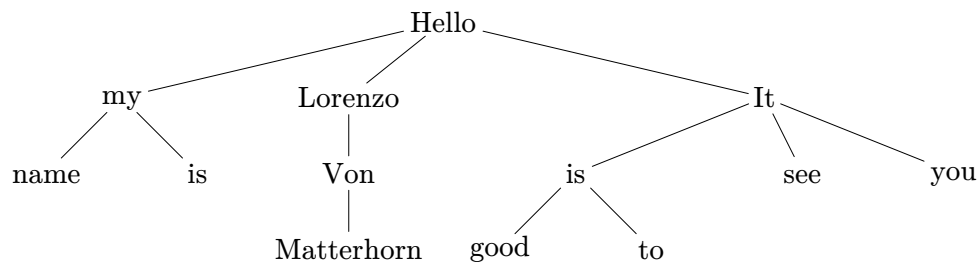
§1.7.1. Examples of Trees

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 -

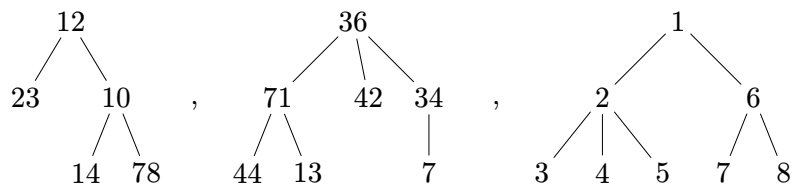


§1.7.2. Making Larger Trees from Smaller Trees

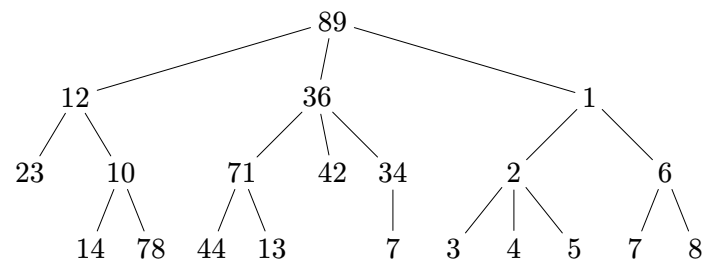
If we have an object -

89

and a few trees -



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



§1.7.3. Formal Definition of Trees

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

≡ tree

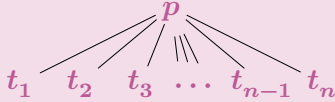
A **tree over a set S** defines a meaningful structure on a collection of elements from S .

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

Given a mathematical object t ,

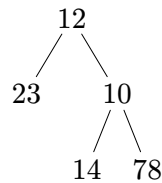
- first check whether $t \in S$, in which case t passes the check, and is a **tree over S**

Failing that,

- check whether t is of the form , where

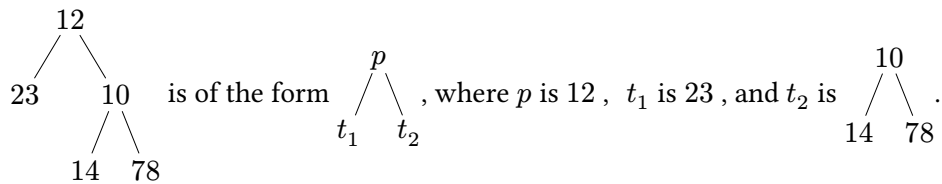
- $p \in S$
- and each of $t_1, t_2, t_3, \dots, t_{n-1}$, and t_n is a **tree over S** .

Let us use this definition to check whether

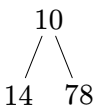


is a **tree over the natural numbers**.

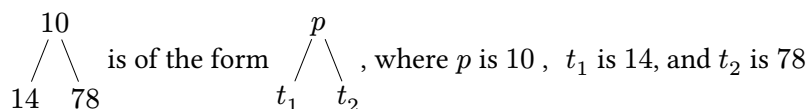
Let's start -



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

So we are only left to check that 23 and  are trees over the natural numbers.

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



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.

§1.7.4. Structural Induction

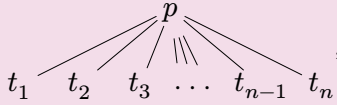
In order to prove things about trees, we have a version of the

≡ **principle of mathematical induction** for trees -

✚ structural induction for trees

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

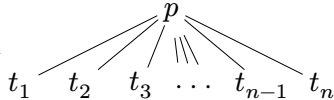
then φ_t is true for each tree t over S .

§1.7.5. 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 ($f : \text{Trees over } S \rightarrow X$), we can divide taking the input into two cases, and define the outputs respectively.

Let's use this principle to define the function

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

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

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

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

§1.7.6. Termination

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

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

Proof For each tree t , let φ_t be the statement

“ The definition of $\text{size}(t)$ will terminate “

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

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

- $\langle\langle \text{For each tree } T \text{ of the form } \dots \text{ then } \varphi_T \text{ is also true} \rangle\rangle$

Assume that each of $\varphi_{t_1}, \varphi_{t_2}, \varphi_{t_3}, \dots, \varphi_{t_{n-1}}, \varphi_{t_n}$ is true.

That means that each of $\text{size}(t_1), \text{size}(t_2), \text{size}(t_3), \dots, \text{size}(t_{n-1}), \text{size}(t_n)$ will terminate.

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

Thus, we can see that each term in the right-hand side terminates.

Therefore, the left-hand side “ $\text{size}(\tau)$ ”,

being defined as a well-defined combination of these terms,
must also terminate.

Hence φ_T is proved!

Hence the theorem is proved!! ■

§1.8. 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.

§1.8.1. 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 $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$, we can find out its value by finding out the values of its inputs and then applying f on these values.

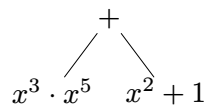
So, for dumb Haskell to do this (figure out the values of expressions, which is quite an important ability), a vital requirement is to be able to easily put expressions in the form $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$.

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.

§1.8.2. The Solution

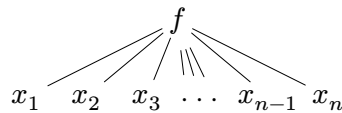
One way to make this easier to represent the expression in the form of a tree -

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

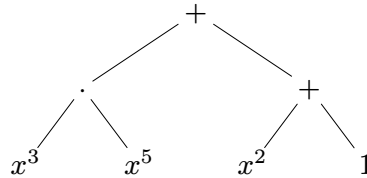


, it becomes obvious what the function is and what the inputs are to which it is applied.

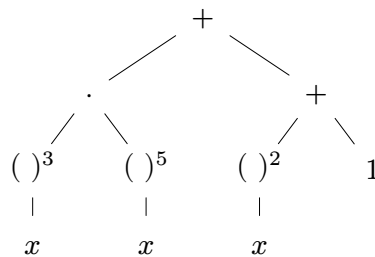
In general, we can represent the expression $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$ as



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

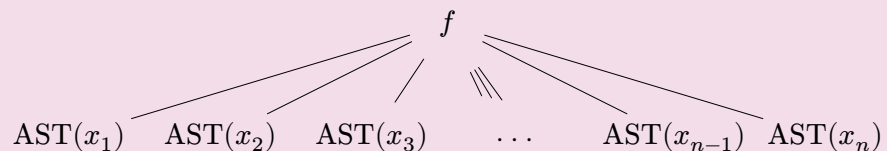
The **abstract syntax tree of a well-formed expression** is defined by applying the “function” **AST** to the expression.

The “function” **AST** is defined as follows -

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

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

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



§1.8.3. Exercises

x Turbo The Snail(IMO 2024,5)

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.

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 Usmons(USA TST 2015, simplified)

A physicist encounters 2015 atoms called usmons. 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 usmons in a line such that all the charged usmons are to the left of the uncharged usmons, regardless of the number of charged usmons. 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 at least 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 at least 2.

x Squarefull

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

x The best (trollest) codeforces question ever!

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 exmample:

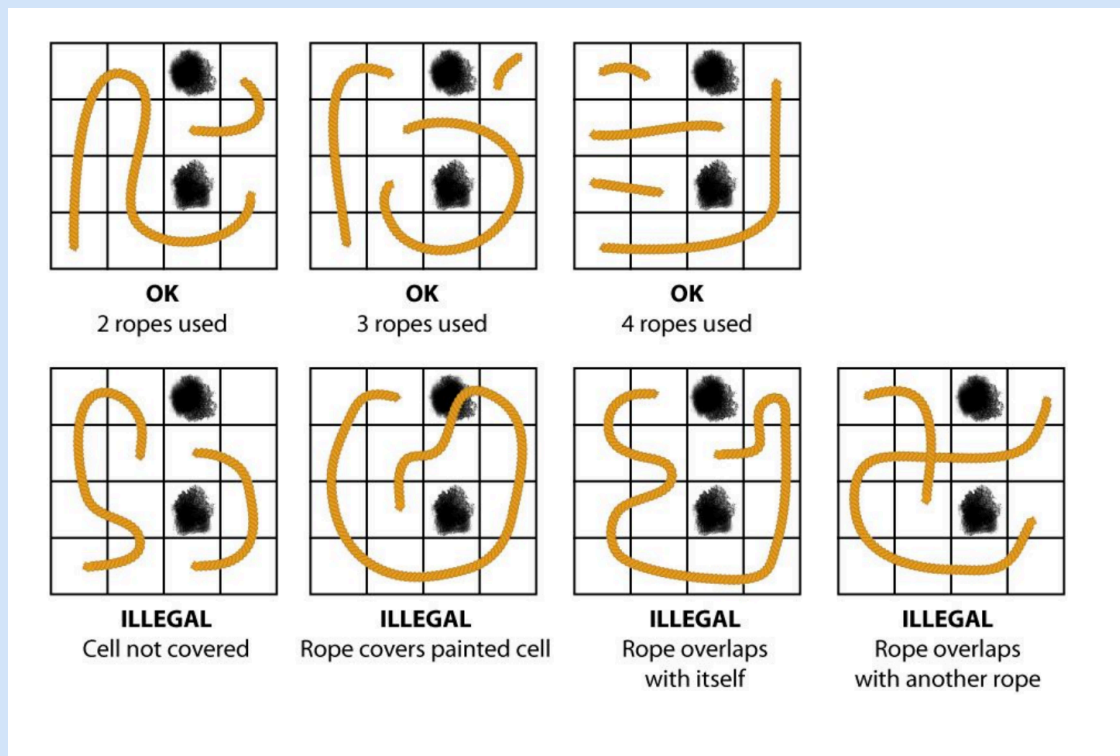


Figure out an algorithm/recipie to covering the grid using $n + 1$ ropes leagally.

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

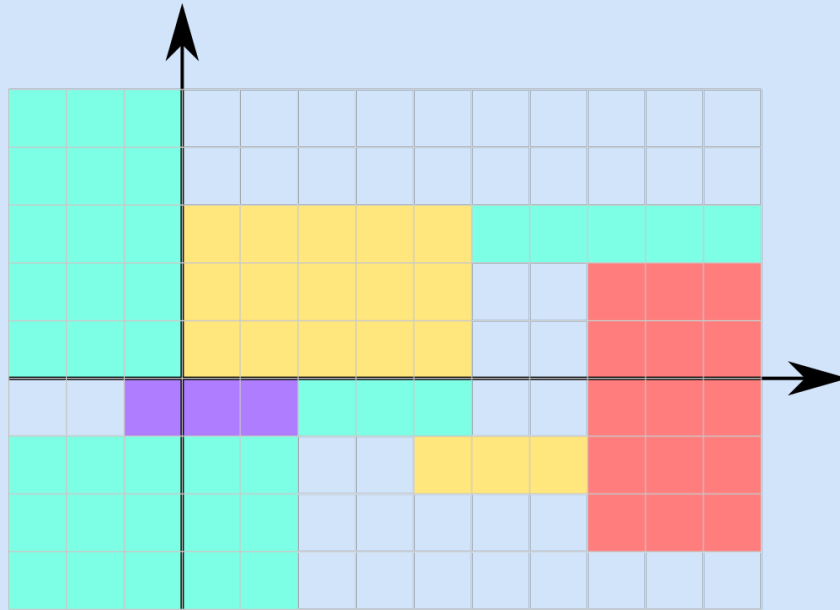
Prove that for every positive integer n , there exists an n -digit number divisible by 5^n , all of whose digits are odd.

x This was rated 2100? (Timofey's Colourbook Problem, Codeforces)

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/recipe), 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 The scenic way

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

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

(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 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

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

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 $P_1, P_2, P_3, \dots, P_n$ in such a way that P_1 defeated P_2 , P_2 defeated P_3 , \dots , P_{n-1} defeated P_n .

X Stamps

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.

X 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 milk they started with?

X 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 $\frac{2 + (-\frac{1}{2})^n}{3}$.

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.

Installing Haskell

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

§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 [haskellSupport/haskell/installation/Linux](#) in your text editor.
(We have more support for Visual Studio Code, but any text editor should do)
4. Open the terminal of your text editor and ensure that current directory is [Linux](#).
5. Type in `installHaskell` in the terminal.
6. This may take a while.
7. You will know installation is complete at the point when it says [Press any key to exit](#).
8. 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 [haskellSupport/haskell/installation/Linux](#) in your text editor.
(We have more support for Visual Studio Code, but any text editor should do)
4. Open the terminal of your text editor and ensure that current directory is [Linux](#).
5. Type in [installHaskellSupport](#) in the terminal.
6. This may take a while.
7. You will know installation is complete at the point when it says [haskellSupport installation complete](#).
8. 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 [haskellSupport](#) in Finder.
4. Open the folder [haskell](#) in Finder .
5. Open the folder [installation](#) in Finder.
6. Right click on the folder [MacOS](#) in Finder, and select [Open in Terminal](#).
7. Type in [chmod +x installHaskell.command](#) in the terminal.
8. Close the terminal window.
9. Open the folder [MacOS](#) in Finder.
10. Double-click on [installHaskell.command](#).
11. This may take a while.
12. You will know installation is complete at the point when it says [Press any key to exit](#).
13. 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 [haskellSupport](#) in Finder.
4. Open the folder [haskell](#) in Finder .
5. Open the folder [installation](#) in Finder.

6. Right click on the folder `MacOS` in Finder, and select `Open in Terminal`.
7. Type in `chmod +x installHaskellSupport.command` in the terminal.
8. Close the terminal window.
9. Open the folder `MacOS` in Finder.
10. Double-click on `installHaskellSupport.command`.
11. This may take a while.
12. You will know installation is complete if a new window pops up with `helloWorld` written in it.
13. 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 `haskellSupport` in File Explorer.
4. Open the folder `haskell` in File Explorer .
5. Open the folder `installation` in File Explorer.
6. Open the folder `Windows` in File Explorer.
7. Double-click on `installHaskell`.
8. This may take a while.
9. You will know installation is complete at the point when it says `Press any key to exit`.
10. 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 `haskellSupport` in File Explorer.
4. Open the folder `haskell` in File Explorer.
5. Open the folder `installation` in File Explorer.
6. Open the folder `Windows` in File Explorer.
7. Double-click on `installHaskellSupport`.
8. This may take a while.

9. You will know installation is complete if a new window pops up with `helloWorld` written in it.
10. Restart (shut down and open again) your device.

Basic Syntax

Arjun Maneesh Agarwal

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

§3.1.1. Introduction to Types

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

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

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

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

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

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

While we recommend, atleast for the early chapters, to declare the types of your functions explicitly ex. `(+) :: Int → Int → Int`; Haskell has a type inference system² which is quite accurate and tries to go for the most general type. This can be both a blessing and curse, as we will see in a few moments.

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

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

`Int` and `Integer` are the types used to represent integers.

`Integer` can hold any number no matter how big, up to the limit of your machine's memory, while `Int` corresponds to the set of positive and negative integers that can be expressed in 32 or 64 bits (based on system) with the bounds changing depending on implementation (guaranteed at least -2^{29} to 2^{29}). Going outside this range may give weird results. Ex. `product [1..52] :: Int` gives a negative number which cannot realistically be 52!. On the other hand, `product [1..52] :: Integer` gives indeed the correct answer.

The reason for `Int` existing despite its bounds and us not using `Integer` for everything is related to speed and memory. Using the former is faster and uses lesser memory.

```
>>> product [1..52] :: Int
-8452693550620999680
(0.02 secs, 87,896 bytes)
>>> product [1..52] :: Integer
80658175170943878571660636856403766975289505440883277824000000000000
(0.02 secs, 123,256 bytes)
```

Almost 1.5 times more memory is used in this case.

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

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

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

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

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

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

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

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

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

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

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

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

§3.1.2. Logical Operations

For example -

Write Haskell code to simulate the following logical operators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

square :: Int → Int
square x = x^2

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

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

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

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

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

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

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


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

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

17 Xors

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

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

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

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

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

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

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

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

17 Xors contd.

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

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

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

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

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

17 Xors, contd.

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

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

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

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

We do this as follows

A 17 Xors, ctd

```

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

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

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

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

xor14 b1 b2
  | b1 == b2 = False
  | b1 /= b2 = True

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

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

```

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

```

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

```

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

A 17 Xors, contd

```

xor16, xor17 :: Bool → Bool → Bool

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

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

```

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

X Exercise

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

§3.1.3. Numerical Functions

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

```

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

```

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

⚠ Implementation of abs function

```

abs :: Num a => a -> a
abs a = if a ≥ 0 then a else -a

```

§3.1.3.1. 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` typeclass. This refers to a lot of things, but the most common ones are `Rational`, `Float` and `Double`. `recip`, as the name suggests, returns the reciprocal of the number taken as input. The type signature is `recip :: Fractional a => a -> a`

```

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

```

It is clear that in the above case, 5 was treated as a `Float` or `Double` and the expected output provided. In the following case, we specified the type to be `Int` and it caused a horrible error. This is because

for something to be a fractional type, we literally need to define how to reciprocate it. We will talk about how exactly it is defined in < some later chapter probably 8 >. For now, once we have `recip` defined, division can be easily defined as

```
(/) :: Fractional a => a -> a -> a
x / y = x * (recip y)
```

Again, notice the type signature of `(/)` is `Fractional a => a -> a -> a`.³

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

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

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

Exercise

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

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

³It 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.

X Exercise

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

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

While the functions work 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 ideally 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 up to 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.

A division algorithm on positive integers by repeated subtraction

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

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

```

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

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

```

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

§3.1.3.2. Exponentiation

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

x Exercise

What can we say about the three exponation operators?

<will make this example later>

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

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

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

```

>>> 2^1000
10715086071862673209484250490600018105614048117055336074 ...
>>> 2 ^^ 1000
1.0715086071862673e301
>>> 2^10000
199506311688075838488374216268358508382 ...
>>> 2^^10000
Infinity
>>> 2 ** 10000
Infinity

```

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

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

⚠ A naive integer exponentiation algorithm

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

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

However, we could do a bit better here. Notice, to evaluate a^b , we are making b multiplications. A fact we mentioned before is that multiplication of big numbers is faster when it is 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.

⚠ A better exponentiation algorithm using divide and conquer

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

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

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

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

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

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

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

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

§3.1.3.3. 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.

A naive way to do so would be:

⚠ Naive GCD and LCM

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

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

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

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

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

Fast GCD and LCM

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

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

We can see that this is much faster. The exact number of steps or time taken is a 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.

§3.1.3.4. Recursive Functions

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

Factorial, Binomial and Fibonacci

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

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

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

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

§3.1.4. 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?

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

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

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

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

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

Here's how it works:

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

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

While using a raw 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:

⚠ Square root by binary search

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

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

The internal implementation sets the tolerance to some constant, defining, for example as

```
sqr = 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.

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

We know that $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$. 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 .

Log defined using Taylor Approximation

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

This is a very efficient algorithm for approximating `log`. Doing better requires the use of either pre-computed lookup tables(which would make the programme heavier) or use more sophisticated mathematical methods which while more accurate would slow the programme 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

Sin and Cos using Taylor Approximation

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

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

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

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

Finally, 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)`.

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

§3.1.5. Exercise

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 .

x 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 $n \rightarrow \infty$, $f(x_n) \rightarrow 0$.

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.

x 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`.

x AGM Log

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

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

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

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

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

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

x Multiplexer

A multiplexer is a hardware element which chooses the input stream from a variety of streams. It is made up of $2^n + n$ components where the 2^n are the input streams and the n are the selectors.

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

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

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

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

x 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 1x1 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.

Types as Sets

Ryan Hota

§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,

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

⊢ 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 $f : A \rightarrow B$* ,
 or equivalently, the *set of all functions from A to B* .

✕ size of exponent set

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

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

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 \vdash 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 $y \in \{\text{False}, \text{True}\}$. Take the value of y to be the \oplus of True and False.”

✕ declaring a variable

Declare a variable of type `Char`.

§4.2.2. $A \rightarrow B$ is analogous to B^A or \Rightarrow 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 $f :: A \rightarrow B$.

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 $\text{Bool} \rightarrow \text{Integer}$.

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 \times 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 a is of type A and b is of type B .

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

- λ type of a pair

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

This reads - "GHCi, what is the type of $(\text{True}, 'c')$?

Answer : the type of $(\text{True}, 'c')$ is $(\text{Bool}, \text{Char})$."

If we have a type X with elements x_1, x_2 , and x_3 , and another type Y with elements y_1 and y_2 , 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 )
```

✕ 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 `()`.”

x function to unit

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

x function from unit

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

§4.2.5. No \div 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 \div union of Types

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

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

x subtype

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

§4.2.7. Disjoint Union of Sets

\div 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,

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

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

For example, consider the statement - $(0, 10) \in \mathbb{R} \sqcup \mathbb{N}$.

It is obvious that this 10 comes from \mathbb{R} and does not come from \mathbb{N} .

$(1, 10) \in \mathbb{R} \sqcup \mathbb{N}$ would indicate exactly the opposite, i.e, the 10 here comes from \mathbb{N} , not \mathbb{R} .

§4.2.8. **Either A B** is analogous to $A \sqcup B$ or \oplus **disjoint union**

The term “either” is motivated by its appearance in the definition of \oplus **disjoint union**.

Recall that in a \oplus **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 -

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 -

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., $(x \mapsto \frac{1}{x}) : \mathbb{Q} \rightarrow \mathbb{Q}$.

Sounds simple enough! Let's see -

λ naive reciprocal

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

But there is a small issue! What about $\frac{1}{0}$?

What should be the output of `reciprocal 0`?

Unfortunately, it results in an error -

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

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

So the new output type would look something like this - $(\{extra\ element\} \sqcup Rational)$

Notice that this $\{extra\ element\}$ is a $\{ \}$ **singleton set**.

Which means that if we take this *extra element* to be the value `()`,

and take $\{extra\ element\}$ to be the type `()`,

then we can obtain $(\{extra\ element\} \sqcup Rational)$ as the type `Either () Rational`.

Then we can finally rewrite 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 its 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 its 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.

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

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`.

Introduction to Lists

Ryan Hota

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

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

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]`.

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

>>> :t +d ['a','z','\STX']
['a','z','\STX'] :: [Char]

>>> :t +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 by writing `[]`.

§5.2.2. Arithmetic Progression

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

```
>>> [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.

```
>>> [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?

§5.4. 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 -

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

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 -

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

,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)]
```

§5.4.1. Cons or `(:)`

The operator `:` (read as “cons”) can be used to add a single element to 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.

§5.5. 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` . “

§5.5.1. 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. “

§5.5.2. 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`.

§5.5.3. 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 -

✕ 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])
```

§5.5.4. Elem

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 -

```
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.”

§5.5.5. (!!)

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.

§5.6. 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!"

>>> :t +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"
```

§5.7. Structural Induction for Lists

Suppose we want to prove some fact about lists.

We can use the following version of the \equiv **principle of mathematical induction** -

\equiv structural induction for lists

Suppose for each list `l` of type `[T]`, we have a statement φ_l . If we can prove 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 us 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 \equiv **structural induction for lists**, we need to prove -

- $\langle\langle \varphi [] \rangle\rangle$

The definition of `length []` directly gives `0`.

- $\langle\langle \text{For each list } (x:xs), \text{ if } \varphi_{xs}, \text{ then } \varphi_{(x:xs)} \text{ also.} \rangle\rangle$

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. ■

§5.8. Optimization

Suppose we want to reverse 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 -

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 [5,8,3,2] = ( 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`

λ **naive splitAt**

```
splitAt n l = ( take n l , drop n l )
```

λ **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 \div **structural induction for lists**.
- (2) See which definition takes more steps to compute `splitAt 2 [5,8,3]`

§5.9. Lists as Syntax Trees

Recall \div **abstract syntax tree**.

Remember that we represent $f(x,y)$ as $\begin{array}{c} f \\ \swarrow \searrow \\ x \quad y \end{array}$

Using this rule, see whether the following steps make sense -

$$[5,8,3] == (:) 5 [8,3]$$

$$== \begin{array}{c} (:) \\ \swarrow \searrow \\ 5 \quad [8,3] \end{array}$$

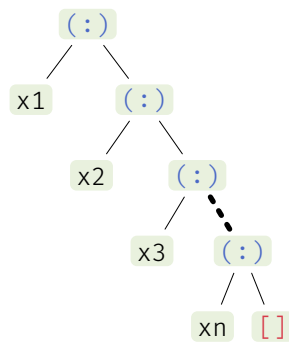
$$== \begin{array}{c} (:) \\ \swarrow \searrow \\ 5 \quad (:) 8 [3] \end{array}$$

$$== \begin{array}{c} (:) \\ \swarrow \searrow \\ 5 \quad (:) \\ \quad \swarrow \searrow \\ \quad 8 \quad [3] \end{array}$$

$$== \begin{array}{c} (:) \\ \swarrow \searrow \\ 5 \quad (:) \\ \quad \swarrow \searrow \\ \quad 8 \quad (:) 3 [] \end{array}$$

$$== \begin{array}{c} (:) \\ \swarrow \searrow \\ 5 \quad (:) \\ \quad \swarrow \searrow \\ \quad 8 \quad (:) \\ \quad \quad \swarrow \searrow \\ \quad \quad 3 \quad [] \end{array}$$

In fact any list `[x1,x2,x3, ..., xn]` can be represented as



This is the representation that Haskell actually uses to store lists.

§5.10. 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 0s -

```
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 `take` to view an initial section of the list, rather than the whole thing.

Polymorphism and Higher Order Functions

Shubh Sharma

§6.1. Polymorphism

§6.1.1. Classification has always been about *shape* and *behaviour* 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 behaviour of elements, that is, the functions are defined on them and how one combine simpler functions defined on an element to define more complicated ones.

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 specified amount of items in the list from the beginning.

These functions seem to not 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 is 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 structure 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 definition 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 elements 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 involves 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.

④ 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 input 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 xs = xs
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 behavior, 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 behaviour 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`), for that we will need a way to group together types by shared *behaviour*, which we will see in the next section.

÷ 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 them have the same output, so mathematicians call them the same, and write $f = g$ this is called **function extensionality**. But 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 between functions, by simply looking at which one works faster or slower on big inputs, and that might be something people might want to factor into what they mean by “sameness”. So maybe the assumption that 2 functions being equal pointwise imply the functions are equal 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 let's 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:

```
λ elem
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem e (x : xs) = e == x || elem e xs
```

✕ 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 is in an ascending order (equal elements are allowed to be next to each other), and false otherwise.

✕ Shape is behaviour?

The two types of polymorphism, that is parametric and ad-hoc, are not exclusive, there are plenty of functions 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 arguments 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 [Chapter 3](#). We have been suggesting to think of the type as

$(+) :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$, because that is really what we want the function to do, but in haskell it would actually mean $(+) :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$, which says the function has 1 interger argument, and it returns a function of type $\text{Int} \rightarrow \text{Int}$.

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

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

Here $(\mathbb{R} \rightarrow \mathbb{R})^d$ 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:

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

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 $(C^B)^A$ such that given any function $f : C^{A \times B}$ we have

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

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

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

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

The inverse of *curry* is called *uncurry* : $(C^B)^A \rightarrow C^{A \times B}$

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

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

Exercise

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

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

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

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

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

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 combine them is fair standard problem solving strategy, in both Mathematics and Computer Science, and in fact in a lot more general scenarios too! Having a clean notation for a tool that used fairly frequently is always a good idea!

Higher order functions are where polymorphism shines it brightest, see how the composition function works on all pairs of functions that can be composed in the mathematical sense, this

would have been significantly less impressive if say it was only composition between functions from `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
```

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

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

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

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

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

which in my opinion is much simpler to read!

✕ Exercise

Write a function `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 example: `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 for an analysis that happens before running the program, this is called **type checking**.

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

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

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

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

§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`

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 indexing 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.

Polymorphism and Higher Order Functions

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 together to form the functions which solve the problem.
- Turns out that one of 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 propagates in this manner through a large section of your code, this means that a huge chunk of code needs to be rewritten to look something like:

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

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

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

This is still not a very elegant way to write things though, and it's 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:

$(a \rightarrow b) \rightarrow (Maybe\ a \rightarrow Maybe\ b)$ and we define it as follows:

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

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

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

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

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

So consider the following chain of functions:

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

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

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

x Beyond map

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

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

```
v :: Maybe a
f :: a → Maybe b

g = f <$> v :: Maybe (Maybe b)
```

This is most likely undesirable, the point of `Maybe` was to say that there is a possibility of error, the point of `maybeMap` was to propagate that possible error then the type `Maybe (Maybe b)` seems to not have a place here, in such cases one can define `maybeJoin :: Maybe (Maybe a) → Maybe a`, with that we can have

```
g = maybeJoin $ f <$> a :: Maybe b
```

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

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

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

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

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

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

Advanced List Operations

Arjun Maneesh Agarwal

§7.1. advanced lists (feel free to change it)

§7.1.1. List Comprehensions

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

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

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

which can be written in Haskell as

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

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

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

⚠ Defining map using pattern matching and list comprehension

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

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

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

⚠ Defining filter using pattern matching and list comprehension

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

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

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

⚠️ **Defining cartesian product using pattern matching and list comprehension**

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

-- Trying to define this recursively is much more cumbersome.

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

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

⚠️ **A naive way to get pythagorean triplets**

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

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

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

⚠️ **A mid way to get pythagorean triplets**

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

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

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

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

λ The optimal way to get pythagorean triplets

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

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

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

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

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

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

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

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

So we can implement `merge`, rather simply as

λ The merge function of mergesort

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

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

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

That means, we can now merge sort using the function

 An implementation of mergesort

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

MergeSort Works?

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

- (i) Prove that `merge` defined by taking the smaller of the heads of the lists recursively, produces a sorted list given the two input lists were sorted. The idea is that the first element chosen has to be the smallest. Use induction of the sum of lengths of the lists.
- (ii) Prove that `mergeSort` works using induction on the size of list to be sorted.

This is also a very efficient way to sort a list. If we define a function C that count the number of comparisons we make, $C(n) < 2 * C(\lceil \frac{n}{2} \rceil) + n$ where the n comes from the merge.

This implies

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

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

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

$$\exists m \text{ s.t. } \forall n > m : n \log(n) > 3n > \frac{1}{2} \log(n) > \frac{1}{2}$$

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

In practice, we waste some amount of operations dividing the list in 2. What if we take our chances and approximately divide the list into two parts?

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

An implementation of Quick Sort

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort [x] = [x]
quickSort (x:xs) = quickSort [l | l <- xs, l > x] ++ [x] ++ quickSort [r |
r <- xs, r <= x]
```

Quick Sort works?

Prove that Quick Sort does indeed work. The simplest way to do this is by induction on length.

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

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

In the worst case scenario, our algorithm could keep splitting the list into a length 0 and a length $n - 1$ list. This would screw us very badly.

As $C(n) = C(0) + C(n - 1) + 2(n - 1)$ where the $n - 1$ comes from the list comprehension and the $(n - 1) + 1$ from the concatenation. Using $C(0) = 0$ as we don't make any comparisons, This evaluates to

$$\begin{aligned} C(n) &= C(n - 1) + 2(n - 1) \\ &= 2(n - 1) + 2(n - 2) + \dots + 2 \\ &= 2 * \frac{n(n - 1)}{2} \\ &= n^2 - n \end{aligned}$$

Which is quite bad as it grows quadratically. Furthermore, the above case is also common enough. How common?

A Strange Proof

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

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

Let's look at the average or expected number of comparison we would need to make!

Consider the list we are sorting a permutation of $[x_1, x_2, \dots, x_n]$. Let $X_{i,j}$ be a random variable which is 1 if the x_i and x_j are compared and 0 otherwise. Let $p_{i,j}$ be the probability that x_i and x_j are compared. Then, $\mathbb{E}(X_{i,j}) = 1 * p + 0 * (1 - p) = p$.

Using the linearity of expectation (remember $\mathbb{E}(\sum X) = \sum \mathbb{E}(x)$), we can say $\mathbb{E}(C(n)) = \sum_{i,j} \mathbb{E}(X_{i,j}) = \sum_{i,j} p_{i,j}$.

Using the same idea we used to reduce the number of pythagoream triplets we need to check, we rewrite this summation as

$$\begin{aligned}\mathbb{E}(C(n)) &= \sum_{i,j} p_{i,j} \\ &= \sum_{i=1}^n \sum_{j=i+1}^n p_{i,j}\end{aligned}$$

Despite a toothy appearance, this is rather easy and elegant way to actually compute $p_{i,j}$.

Notice that each element in the array (except the pivot) is compared only to the pivot at each level of the recurrence. To compute $p_{i,j}$, we shift our focus to the elements $[x_i, x_{i+1}, \dots, x_j]$. If this is split into two parts, x_i and x_j can no longer be compared. Hence, x_i and x_j are compared only when from the first pivot from the range $[x_i, x_{i+1}, \dots, x_j]$ is either x_i or x_j .

This clearly has probability $p_{i,j} = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$. Thus,

$$\begin{aligned}\mathbb{E}(C(n)) &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^n 2 \left(\frac{1}{2} + \dots + \frac{1}{n-i+1} \right) \\ &= 2 \sum_{i=1}^n \left(1 + \frac{1}{2} + \dots + \frac{1}{n-i+1} - 1 \right) \\ &\leq 2 \sum_{i=1}^n \log(i) \\ &\leq 2 \sum_{i=1}^n \log(n) \\ &\leq 2n \log(n)\end{aligned}$$

Considering the number of cases where the comparisons with $n^2 - n$ operations is 2^{n-1} , Quick Sort's expected number of operations is still less than $2n \log(n)$ which, as we discussed, is optimal.

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

Haskell has an inbuilt `sort` function you can use by putting `import Data.List` at the top of your code. This used to use quickSort as the default but in 2002, Ian Lynagh changed it to Merge Sort. This was motivated by the fact that Merge Sort gurentees sorting in $n \log(n) + \dots$ comparisons while Quick Sort will sometimes finish much quicker (pun not intended) and other times, just suffer.

As a dinal remark, our implementation of the Quick Sort is not the most optimal as we go through the list twice, but it is the most aesthetically pleasing and concise.

x Faster Quick Sort

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

Try to figure out this implementation.

§7.1.2. Zip it up!

Have you ever suffered through a conversation with a very dry person with the goal of getting the contact information of a person you are actually interested in? If you haven't well, that is what you will have to do now.

x The boring zip

Haskell has an inbuilt function called `zip`. It's behaviour is as follows

```
>>> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
>>> zip [1,2,3] [4,5,6,7]
[(1,4),(2,5),(3,6)]
>>> zip [0,1,2,3] [4,5,6]
[(0,4),(1,5),(2,6)]
>>> zip [0,1,2,3] [True, False, True, False]
[(0,True),(1,False),(2,True),(3,False)]
>>> zip [True, False, True, False] "abcd"
[(True,'a'),(False,'b'),(True,'c'),(False,'d')]
>>> zip [1,3..] [2,4..]
[(1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(13,14),(15,16),(17,18),
(19,20) ... ]
```

What is the type signature of `zip`? How would one implement `zip`?

The solution to the above exercise is, rather simply:

λ Implementation of zip function

```
zip :: [a] → [b] → [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

While one could think of some places where this is useful, all of the uses seem rather dry. But now that `zip` has opened up to us, we will ask them about `zipWith`. The function `zipWith` takes two lists, a binary function, and joins the lists using the function. The possible implementations are:

λ Implementation of zipWith function

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

x Alternate definitions

While we have defined `zip` and `zipWith` independently here, can you: (i) Define `zip` using `zipWith`? (ii) Define `zipWith` using `zip`?

Now one might feel there is nothing special about `zipWith` as well, but they would be wrong. First, it saves us from defining a lot of things: `zipWith (+) [0,2,5] [1,3,3] = [1,5,8]` is a common enough use. And then, it leads to a lot of absolutely mindblowing pieces of code.

λ The `zipWith fibonacci`

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

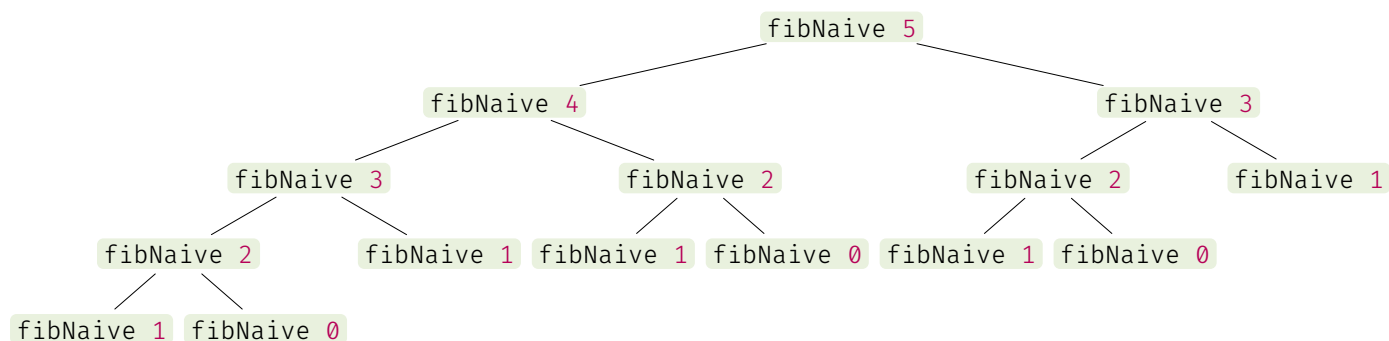
Believe it or not, this should output the fibonacci sequence. The idea is that Haskell is lazy! This means lists are computed one element at a time, starting from the first. Tracing the computation of the elements of `fibs`:

1. Since by definition `fibs = 0 : 1 : (something)`, the first element is `0`.
2. This is again easy, since `fibs = 0 : 1 : (something)`, so the second element is `1`.
3. This is going to be the first element of `something`, i.e. the part that comes after the `0 : 1 :`. So, we need to compute the first element of `zipWith (+) fibs (tail fibs)`. How do we do this? We compute the first element of `fibs` and the first element of `tail fibs` and add them. We know already, that the first element of `fibs` is `0`. And we also know that the first element of `tail fibs` is the second element of `fibs`, which is `1`. So, the first element of `zipWith (+) fibs (tail fibs)` is `0 + 1 = 1`.
4. It is going to be the fourth element of `fibs` and the second of `zipWith (+) fibs (tail fibs)`. Again, we do this by taking the second elements of `fibs` and `tail fibs` and adding them together. We know that the second element of `fibs` is `1`. The second element of `tail fibs` is the third element of `fibs`. But we just computed the third element of `fibs`, so we know it is `1`. Adding them together we get that the fourth element of `fibs` is `1 + 1 = 2`.

This goes on and on to generate the fibonacci sequence. To recall, the naive

```
fibNaive 0 = 0
fibNaive 1 = 1
fibNaive n = fibNaive (n-1) + fibNaive (n-2)
```

is much slower. This is because the computation tree for say `fib 5` looks something like:



x Exercise

Try to trace the computation of `fib !! 5` and make a tree.

Let's now try using this trick to solve some harder problems.

x Tromino's Pizza I

Tromino's sells slices of pizza in only boxes of 3 pieces or boxes of 5 pieces. You can only buy a whole number of such boxes. Therefore it is impossible to buy exactly 2 pieces, or exactly 4 pieces, etc. Create list `possiblePizza` such that if we can buy exactly `n` slices, `possiblePizza !! n` is `True` and `False` otherwise.

The solution revolves around the fact $f(x) = f(x - 3) \vee f(x - 5)$. A naive implementation could be

```
possiblePizzaGo :: Int → Bool
possiblePizzaGo x
  | x == 0    = True
  | x < 3     = False
  | x == 5    = True
  | otherwise = possiblePizzaGo (x - 3) || possiblePizzaGo (x - 5)

possiblePizza = [possiblePizzaGo x | x <- [0..]]
```

This is slow for the same reason as `fibNaive`. So what can we do? Well, use `zipWith`.

```
possiblePizza = True : False : False : True : False : zipWith (||)
  (possiblePizza) (drop 3 possiblePizza)
```

Note, we need to define till the 5th place as otherwise the code has no way to know we can do 5 slices.

x Tromino's Pizza II

Tromino's has started to charge a box fees. So now given a number of slices, we want to know the minimum number of boxes we can achieve the order in. Create a list `minBoxPizza` such that if we can buy exactly `n` slices, the list displays `Just` the minimum number of boxes the order can be achieved in, and `Nothing` otherwise. The list is hence of type `[Maybe Int]`.

Hint : Create a helper function to use with the `zipWith` expression.

One more interesting thing we can talk about is higher dimensional `zip` and `zipWith`. One way to talk about them is `zip3 :: [a] → [b] → [c] → [(a,b,c)]` and `zipWith3 :: (a → b → c → d) → [a] → [b] → [c] → [d]`. These are defined exactly how might expect them to be.

```

zip3 :: [a] → [b] → [c] → [(a,b,c)]
zip3 [] _ _ = []
zip3 _ [] _ = []
zip3 _ _ [] = []
zip3 (x:xs) (y:ys) (z:zs) = (x,y,z) : zip3 xs ys zs

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

```

x Exercise

A slightly tiresome exercise, try to define `zip4` and `zipWith4` blind.

Haskell predefines till `zip7` and `zipWith7`. We are yet to see anything beyond `zipWith3` used in code, so this is more than enough. Also, if you truly need it, `zip8` and `zipWith8` are not that hard to define.

x Tromino's Pizza III

Tromino's has introduced a new box of size 7 slices. Now they sell 3, 5, 7 slice boxes. They still charge the box fees. So now given a number of slices, we still want to know the minimum number of boxes we can achieve the order in. Create a list `minBoxPizza` such that if we can buy exactly `n` slices, the list displays `Just` the minimum number of boxes the order can be achieved in, and `Nothing` otherwise. The list is hence of type `[Maybe Int]`.

Another idea of dimension would be something that could join together two grids, something with type signature `zip2d :: [[a]] → [[b]] → [[(a,b)]]` and `zipWith2d :: (a → b → c) → [[a]] → [[b]] → [[c]]`.

```

zip2d :: [[a]] → [[b]] → [[(a,b)]]
zip2d = map zip

zipWith2d :: (a → b → c) → [[a]] → [[b]] → [[c]]
zipWith2d = zipWith . zipWith

```

The second definition should raise immediate alarms. It seems too good to be true. Let's formally check

```

zipWith . zipWith $ (a → b → c) [[a]] [[b]]
= zipWith (zipWith (a → b → c)) [[a]] [[b]] -- Using the fact that
-- composition only allows one of the inputs to be pulled inside
= [
    zipWith (a → b → c) [a1] [b1],
    zipWith (a → b → c) [a2] [b2],
    ...
]
= [[c1], [c2], ...]
= [[c]]

```


This also implies `zip2d = zipWith.zipWith $ (\x y → (x,y))` is also a correct definition. Also surprisingly, `zipWith . zipWith . zipWith` has the type signature `(a → b → c) → [[[a]]] → [[[b]]] → [[[c]]]`. You can see where we are going with this...

x Composing zipWith's

What should the type signature and behaviour of `zipWith . zipWith . <n times> . zipWith` be? Prove it.

x Unzip

Haskell has an inbuilt function called `unzip :: [(a,b)] → ([a],[b])` which takes a list of pairs and provides a pair of list in the manner inverse of `zip`.

Try to figure out the implementation of `unzip`.

§7.1.3. Folding, Scanning and The Gate to True Powers

§7.1.3.1. Orgami of Code!

A lot of recursion on lists has the following structure

```
g [] = v -- The vacous case
g (x:xs) = x `f` (g xs)
```

That is, the function `g :: [a] → b` maps the empty list to a value `v`, of say type `b`, and for non-empty lists, the head of the list and the result of recursively processing the tail are combined using a function or operator `f :: a → b → b`.

Some commone examples from the inbuilt functions are:

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

product :: [Int] → Int
product [] = 1 -- The sturcuture forces this choice as other wise, the
product of full lists may become incorrect.
product (x:xs) = x * (product xs)

or :: [Bool] → Bool
or [] = False -- As the structure of our implementation forces this to be
false, or otherwise, everything is true.
or (x:xs) = x || (or xs)

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

We will also see a few more examples in a while, but one can notice that this is a common enough pattern. So what do we do? We abstract it.

A Definition of foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v [] = v
foldr f v (x:xs) = x `f` (foldr f v xs)
```

This shortens our definitions to

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
```

Sometimes, we don't wish to define a base case or maybe the logic makes it so that doing so is not possible, then you use `foldr1 :: (a → b → b) → [a] → a` defined as

A Definition of foldr1

```
foldr1 :: (a → a → a) → [a] → a
foldr1 _ [x] = x
foldr1 f (x:xs) = x `f` (foldr1 f xs)
```

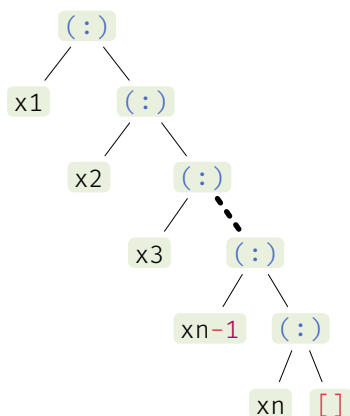
Like we could now define `product = foldr1 (*)` which is much more clean as we don't have to define a weird vacuous case.

Let's now discuss the naming of the pattern. Recall, `[1,2,3,4]` is syntactic sugar for `1 : 2 : 3 : 4 : []`. We are just allowed to write the former as it is more aesthetic and convenient. One could immediately see that

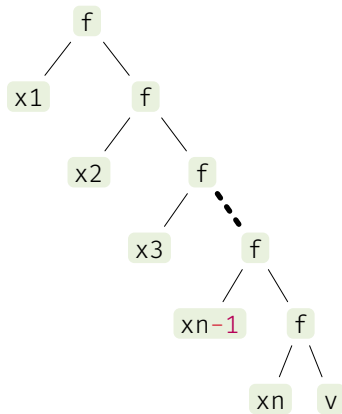
```
foldr v f [1,2,3,4] = 1 `f` (2 `f` (3 `f` (4 `f` v)))
-- and if f is right associative
= 1 `f` 2 `f` 3 `f` 4 `f` v
```

We have basically changed the cons (`:`) into the function and the empty list (`[]`) into `v`. But notice the brackets, the evaluation is going from right to left.

Using trees, a list can be represented in the form



which is converted to:



However, what if our function is left associative? After all, if this was the only option, we would have called it `fold`, not `foldr` right?

The recursive pattern

```
g :: (b -> a -> b) -> b -> [a] -> b
g _ v [] = v
g f v (x:xs) = g (f v x) xs
```

is abstracted to `foldl` and `foldl1` respectively.

λ Definition of `foldl` and `foldl1`

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v [] = v
foldl f v (x:xs) = foldl (f v x) xs

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

And as the functions we saw were commutative, we could define them as

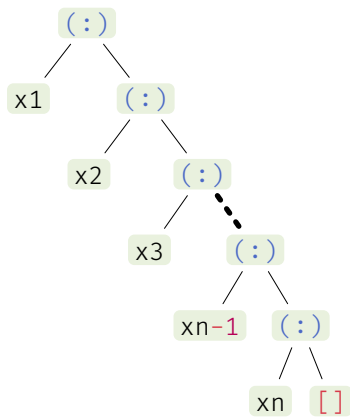
```
sum      = foldl (+) 0
product  = foldl (*) 1
or       = foldl (||) False
and      = foldl (&&) True
```

There is one another pair of function defined in the fold family called `foldl'` and `foldl1'` which are faster than `foldl` and `foldl1` and don't require a lot of working memory. This makes them the defaults used in most production code, but to understand them well, we need to discuss how haskell's lazy computation actually works and is there a way to bypass it. This is done in chapter 9. We will use `foldl` and `foldl1` till then.

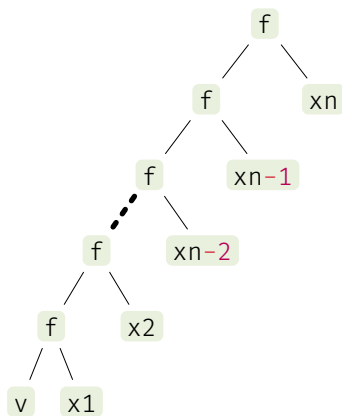
The computation of `foldl` proceeds like

```
foldl v f [1,2,3,4] = (((((v `f` 1) `f` 2) `f` 3) `f` 4)
-- and if f is left associative
= v `f` 1 `f` 2 `f` 3 `f` 4
```

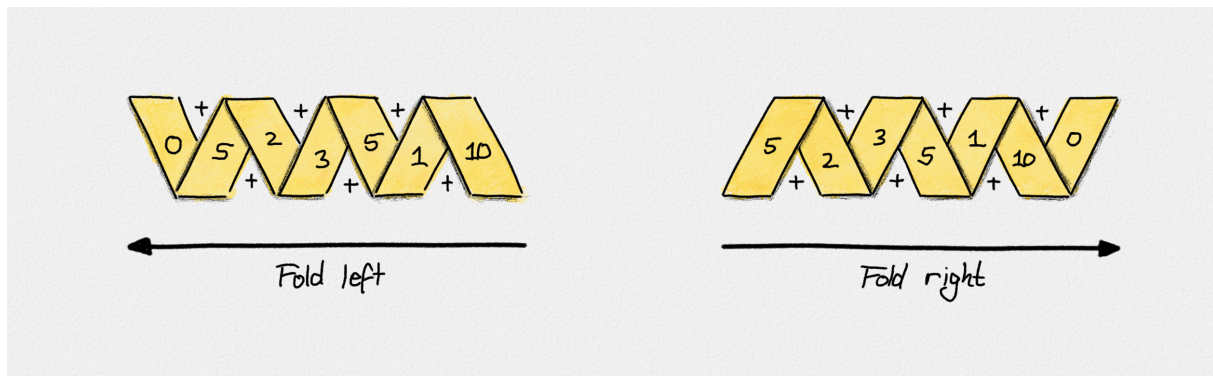
Or in tree form as



which is converted to:



Another very cute picture to summarize the differences is:



Similar to how `unzip` was for `zip`, could we define `unfoldr`, something that takes a generator function and a seed value and generates a list out of it.

What could the type of such a function be? Well, like with every design problem; let's see what our requirements are:

- The list should not just be one element over and over. Thus, we need to be able to update the seed after every unfolding.
- There should be a way for the list to terminate.

So what could the type be? We can say that our function must spit out pairs: of the new seed value and the element to add to the list. But what about the second condition?

Well, what if we can spit out some seed which can never come otherwise and use that to signal it? The issue is, that would mean the definition of the function has to change from type to type.

Instead, can we use something we studied in ch-6? Maybe.⁴

λ Implementation of unfoldr

```
unfoldr :: (a → Maybe (a,b)) → a → [b]
unfoldr gen seed = go seed
  where
    go seed = case gen seed of
      Just (x, newSeed) → x : go newSeed
      Nothing           → []
```

For example, we could now define some library functions as:

```
replicate :: Int → a → [a]
-- replicate's an value n times
replicate n x = unfoldr gen n x where
  rep 0 = Nothing
  rep m = Just (x, m - 1)

iterate :: (a → a) → a → [a]
-- given a function f and some starting value x
-- outputs the infinite list [x, f x, f f x, ...]
iterate f seed = unfold (\x → Just (x, f x)) seed
```

While `foldr` and `foldl` are some of the most common favorite function of haskell programmers; `unfoldr` remains mostly ignored. It is so ignored that to get the inbuilt version, one has to `import Data.List`. We will soon see an egregious case where Haskell's own website ignored it. One of the paper we referred was literally titled "The Under-Appreciated Unfold".

× Some more inbuilt functions

Implement the following functions using fold and unfold.

(i) `concat :: [[a]] → [a]` concatenates a list of lists into a single list. For example:

```
concat [[1,2,3],[4,5,6],[7,8],[],[10]] = [1,2,3,4,5,6,7,8,9, 10]
```

(ii) `cycle :: [a] → [a]` cycles through the list endlessly. For example:

```
cycle [2, 3, 6, 18] = [2, 3, 6, 18, 2, 3, ...]
```

(iii) `filter :: (a → Bool) → [a] → [a]` takes a predicate and a list and filters out the elements satisfying that predicate.

(iv) `concatMap :: (a → [b]) → [a] → [b]` maps a function over all the elements of a list and concatenate the resulting lists. Do not use `map` in your definition.

(v) `length :: [a] → Int` gives the number of elements in the provided list. Use `foldr` or `foldl`.

⁴Pun intended.

X Base Conversion

(i) Convert list of digits in base `k` to a number. That is `lis2num :: Int → [Int] → Int` with the usage `lis2num base [digits]`.

(ii) Given a number in base 10, convert to a list of digits in base `k`. `num2lis :: Int → Int → [Int]` with the usage `num2list base numberInBase10`

Let's go part by part. The idea of the first question is simply to understand that `[4,2,3]` in base `k` represents $4 * k^2 + 2 * k + 3 * k^0 = ((0 * k + 4) * k + 2) * k + 3$; doesn't this smell like `foldl`?

```
lis2num :: [Int] → Int → Int
lis2num k = foldl (\x y → k * x + y) 0
```

For part two, the idea is that we can base convert using repeated division. That is,

```
423 `divMod` 10 = (42, 3)
42 `divMod` 10 = (4, 2)
4 `divMod` 10 = (0, 4)
```

It is clear that we terminate when the quotient reaches 0 and then just take the remainders. Does this sound like `unfoldr`?

```
num2lis :: Int → Int → [Int]
num2lis k = reverse . unfoldr gen where
  gen 0 = Nothing
  gen x = Just $ (x `mod` k, x `div` k)
```

X A list of Primes

This is the time when Haskell itself forgot that the `unfoldr` function exists. The website offers the following method to make a list of primes in Haskell as an advertisement for the language.

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Understand this code (write a para explaining exactly what is happening!) and try to define a shorter (and more aesthetic) version using `unfoldr`.

The answer is literally doing what one would do on paper. Like describing it would be a disservice to the code.

A list of primes using unfoldr

```
sieve (x:xs) = Just (x, filter (\y → y `mod` x /= 0) xs)
primes = unfoldr sieve [2..]
```

X Subsequences

Write a function `sublists :: [a] → [[a]]` which takes a list and returns a list of sublists of the given list. For example: `sublists "abc" = "", "a", "b", "ab", "c", "ac", "bc", "abc"` and `sublists [24, 24] = [[], [24], [24], [24, 24]]`.

Try to use the fact that a sublist either contains an element or not. Second, the fact that sublists correspond nicely to binary numerals may also help.

Your function must be compatible with infinite lists, that is take `10 $ sublists [1..] = [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3], [4], [1,4]]` should work.

Please fill in the blanks below

A naive, non-infinite compatible definition is:

```
sublists [] = _____
sublists (x:xs) = concatMap (\ys → _____) (sublists xs)
```

On an infinite list, this definition gets stuck in a non-productive loop because we must traverse the entire list before it returns anything.

Note that on finite cases, the first sublist returned is always `_____`. This means we can state this as `sublists xs = _____ : _____ (sublists xs)`. It is sensible to extend this equality to the infinite case, due to the analogy of `_____`.

By making this substitution, we produce the definition that can handle infinite lists, from which we can calculate a definition that's more aesthetically pleasing and slightly more efficient:

```
_____ -- Base case
sublists (x:xs) = _____ : _____ : concatMap (\ys → _____) (tail .
sublists xs)
```

We can clean this definition up by calculating definitions for `tail.sublists x` and renaming it something like `nonEmpties`. We start by applying `tail` to both sides of the two cases. `nonEmpties [] = tail.sublists [] = _____` and `nonEmpties (x:xs) = tail.sublists (x:xs) = _____`

Substituting all this through the definition.

λ Space to write the definition of sublists

This function can be called in Haskell through the `subsequences` function one gets on importing `Data.Lists`. Our definition is the most efficient and is what is used internally.

Finally, a question which would require you to use a lot of functions we just defined:

x The Recap Problem (Euler's Project 268)

It can be verified that there are 23 positive integers less than 1000 that are divisible by at least four distinct primes less than 100.

Find how many positive integers less than 10^{16} are divisible by at least four distinct primes less than 100.

Hint : Think about PIE but not π .

Something we mentioned was that `foldr` and `unfoldr` are inverse (or more accurately dual) of each other. But their types seem so different. How do we reconcile this?

$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ &\cong (a \times b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ &\cong (a \times b \rightarrow b) \rightarrow (1 \rightarrow b) \rightarrow [a] \rightarrow b \\ &\cong (a \times b \cup 1 \rightarrow b) \rightarrow [a] \rightarrow b \\ &\cong (\text{Maybe}(a, b) \rightarrow b) \rightarrow [a] \rightarrow b \end{aligned}$$

$$\text{Notice, } \text{unfoldr} :: (b \rightarrow \text{Maybe}(a, b)) \rightarrow b \rightarrow [a]$$

And now the duality emerges. $(\text{foldr } f)^{-1} = \text{unfoldr } f^{-1}$.

Some more ideas on the nature of fold can be found in the upcoming chapters on datatypes as well as in the appendix.

Another type of function we sometimes want to define are:

```
sumlength :: [Int] -> (Int, Int)
sumlength xs = (sum xs, length xs)
```

This is bad as we traverse the list twice. We could do this twice as fast using

```
sumlength :: [Int] -> (Int, Int)
sumlength = foldr (\x (a,b) -> (a+x, b + 1)) (0,0)
```

This might seem simple enough, but this idea can be taken to a different level rather immediately.

x Ackerman Function

The Ackerman function is defined as follows:

```
ack :: [Int] -> [Int] -> [Int]
ack [] ys = 1 : ys
ack (x : xs) [] = ack xs [1]
ack (x : xs) (y : ys) = ack xs (ack (x : xs) ys)
```

Define this in one single line using `foldr`.

Let's say $\text{foldr } f \ v \cong \text{ack}$.

```
=> ack [] = v
=> ack (x:xs) = f x (ack xs)
```


This means $v = (1:)$. unfortunately, figuring out f seems out of reach. Luckily, we are yet to use all the information the function provides.

Let's say $\text{foldr } g \ w \cong \text{ack } (x:xs)$.

```
⇒ ack (x:xs) [] = w
⇒ ack (x:xs) (y:ys) = g y (ack (x:xs) ys)
```

This means $w = \text{ack } xs \ [1]$ and

```
ack (x:xs) (y:ys)
= g y (ack (x:xs) ys) ⇔ ack xs (ack (x : xs) ys)
(canceling on both sides)
⇒ g y = ack xs
⇒ g = (\y z → ack xs z)
```

Thus, $g = (\backslash y \ z \rightarrow \text{ack } xs \ z)$.

And finally, now working towards f , we get

```
ack (x:xs)
= f x (ack xs) ⇔ foldr (\y z → ack xs z) (ack xs [1])
(substitution of a = ack xs)
⇒ f x a ⇔ foldr (\y z → a z) (a [1])
⇒ f = (\x a → foldr (\y z → a z) (a [1]))
```

This gives us the definition

```
ack :: [Int] → [Int] → [Int]
ack = foldr (\x a → foldr (\y z → a z) (a [1])) (1:)
```

This might seem like a rather messy definition, but from a theoretical point of view, even this has its importance. The main thing is that folding is faster than recursion at runtime so if no additional overhead is there, folds will run faster.

It is possible, but out of the scope of our current undertaking, to prove that all primitive recursive functions can be written as folds. What does primitive recursive functions mean? Well, that is left for your curiosity.

x Removing duplicates

Haskell has inbuilt function `nub :: Eq a ⇒ [a] → [a]` which is used to remove duplicates in a list. Write a recursive definition of `nub` and then write a definition using folds.

Haskell also has an inbuilt function `nubBy :: (a → a → Bool) → [a] → [a]` which is used to remove elements who report true to some property. That is `nubBy (\x y → x + y == 4) [1,2,3,4,2, 0] = [1,2,4]` as $1+3 = 4$, $2+2 = 4$, $4 + 0 = 4$. Write a recursive definition of `nubBy` and then write a definition using folds.

x More dropping and more taking

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

and `takeWhile :: (a -> Bool) -> [a] -> [a]` take a predicate and a list and drop all elements while the predicate is satisfied and take all objects while the predicate is satisfied respectively.

Implement them using recursion and then using folds.

§7.1.3.2. Numerical Integration

To quickly revise all the things we just learnt, we will try to write our first big-boy code.

Let's talk about numerical Integration. Numerical Integration refers to finding the value of integral of a function, given the limits. This is also a part of the mathematical computing we first studied in chapter 3. To get going, a very naive idea would be:

```
easyIntegrate :: (Float -> Float) -> Float -> Float -> Float
easyIntegrate f a b = (f b + f a) * (b-a) / 2
```

This is quite inaccurate unless `a` and `b` are close. We can be better by simply dividing the integral in two parts, ie $\int_a^b f(x)dx = \int_a^m f(x)dx + \int_m^b f(x)dx$ where $a < m < b$ and approximate these parts. Given the error term is smaller in these parts than that of the full integral, we would be done. We can make a sequence converging to the integral we are interested in as:

Naive Integration

```
integrate :: (Float -> Float) -> Float -> Float -> [Float]
integrate f a b = (easyIntegrate f a b) : zipWith (+) (integrate f a m)
(integrate f m b) where m = (a+b)/2
```

If you are of the kind of person who likes to optimize, you can see a very simple inoptimality here. We are computing `f m` far too many times. Considering, `f` might be slow in itself, this seems like a bad idea. What do we do then? Well, ditch the aesthetic for speed and make the naive integrate as:

Naive Integration without repeated computation

```
integrate f a b = go f a b (f a) (f b)

integ f a b fa fb = ((fa + fb) * (b-a)/2) : zipWith (+) (integ f a m fa
fm) (integ f m b fm fb) where
  m = (a + b)/2
  fm = f m
```

This process is unfortunately rather slow to converge for a lot of functions. Let's call in some backup from math then.

The elements of the sequence can be expressed as the correct answer plus some error term, ie $a_i = A + E$. This error term is roughly proportional to some power of the separation between the limits evaluated (ie $(b - a)$, $\frac{b-a}{2}$...) (the proof follows from Taylor expansion of f . You are recommended to prove the same). Thus,

$$\begin{aligned}
a_i &= A + B \times \left(\frac{b-a}{2^i} \right)^n \\
a_{i+1} &= A + B \times \left(\frac{b-a}{2^{i+1}} \right)^n \\
\Rightarrow a_{i+1} - \frac{1}{2^n} a_i &= A \left(1 - \frac{1}{2^n} \right) \\
\Rightarrow A &= \frac{2^n \times a_{i+1} - a_i}{2^n - 1}
\end{aligned}$$

This means we can improve our sequence by eliminating the error

```
elimerror :: Int -> [Float] -> [Float]
elimerror n (x:y:xs) = (2^^n * y - x) / (2^^n - 1) : elimerror n (y:xs)
```

However, we have now found a new problem. How in the world do we get `n`?

$$\begin{aligned}
a_i &= A + B \times \left(\frac{b-a}{2^i} \right)^n \\
a_{i+1} &= A + B \times \left(\frac{b-a}{2^{i+1}} \right)^n \\
a_{i+2} &= A + B \times \left(\frac{b-a}{2^{i+2}} \right)^n \\
\Rightarrow a_i - a_{i+1} &= B \times \left(\frac{b-a}{2^i} \right)^n \times \left(1 - \frac{1}{2^n} \right) \\
\Rightarrow a_{i+1} - a_{i+2} &= B \times \left(\frac{b-a}{2^{i+1}} \right)^n \times \left(\frac{1}{2^n} - \frac{1}{4^n} \right) \\
\Rightarrow \frac{a_i - a_{i+1}}{a_{i+1} - a_{i+2}} &= \frac{4^n - 2^n}{2^n - 1} = \frac{2^n(2^n - 1)}{2^n - 1} = 2^n \\
\Rightarrow n &= \log_2 \left(\frac{a_i - a_{i+1}}{a_{i+1} - a_{i+2}} \right)
\end{aligned}$$

Thus, we can estimate `n` using the function `order`. We will be using the in-built function `round :: (RealFrac a, Integral b) -> a -> b` in doing so. In our case, `round :: Float -> Int`.

```
order :: [Float] -> Int
order (x:y:z:xs) = round $ logBase 2 $ (x-y)/(y-z)
```

This allows us to improve our sequence

```
improve :: [Float] -> [Float]
improve xs = elimerror (order xs) xs
```

One could make a very fast converging sequence as say `improve $ improve $ improve $ integrate f a b`.

But based on the underlying function, the number of `improve` may differ.

So what do we do? We make an extremely clever move to define a super sequence `super` as

```
super :: [Float] → [Float]
super xs = map (!! 2) (iterate improve xs) -- remeber itterate from the
exercises above?
```

I will re-instate, the implementation of `super` is extremely clever. We are recursively getting a sequence of more and more improved sequences of approximations and constructs a new sequence of approximations by taking the second term from each of the improved sequences. It turns out that the second one is the best one to take. It is more accurate than the first and doesn't require any extra work to compute. Anything further, requires more computations to compute.

Finally, to complete our job, we define a function to choose the term upto some error.

```
within :: Float → [Float] → Float
within error (x:y:xs)
  | abs(x-y) < error = y
  | otherwise = within error (y:xs)
```

ⓧ An optimized function for numerical integration

```
ans :: (Float → Float) → Float → Float → Float → Float → Float
ans f a b error = within error $ super $ integrate f a b
```

With this we are done!

ⓧ Simpson's Rule

Here we have used the approximation $\int_a^b f(x) dx = (f(a) + f(b)) \frac{b-a}{2}$ and used divide and conquer. This is called the Trapezoidal Rule in Numerical Analysis.

A better approximation is called the Simpson's (First) Rule.

$$\int_a^b f(x) dx = \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

Modify the code to now use Simpson's Rule. Furthermore, show that this approximation makes sense (the idea is to find a quadratic polynomial which takes the same value as our function at a , $\frac{a+b}{2}$ and b and using its area).

§7.1.3.3. Time to Scan

We will now talk about folds lesser known cousing scans.

Scans

While `fold` takes a list and compresses it to a single value, `scan` takes a list and makes a list of the partial compressions. Basically,

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f v [x1, x2, x3, x4]
= [
    foldr f v [x1, x2, x3, x4],
    foldr f v [x2, x3, x4],
    foldr f v [x3, x4],
    foldr f v [x4],
    foldr f v []
]
= [
    x1 `f` x2 `f` x3 `f` x4 `f` v,
    x2 `f` x3 `f` x4 `f` v,
    x3 `f` x4 `f` v,
    x4 `f` v,
    v
]
```

and very much similarly as

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl f v [x1, x2, x3, x4]
= [
    foldl f v [],
    foldr f v [x1],
    foldr f v [x1, x2],
    foldr f v [x1, x2, x3],
    foldr f v [x1, x2, x3, x4]
]
= [
    v,
    v `f` x1,
    v `f` x1 `f` x2,
    v `f` x1 `f` x2 `f` x3,
    v `f` x1 `f` x2 `f` x3 `f` x4,
]
```

There are also very much similar `scanr1` and `scanl1`.⁵

The reason the naming is as the internal implementation of these functions look like similar to the definition of the fold they borrow their name from.

⁵Similar to our note in fold, there is a function pair `scanl'` and `scanl1'`, which similar to `foldl'` and `foldl1'`, and have the same set of benefits. This makes them the defaults, but similarly, to understand them well, we need to discuss how haskell's lazy computation actually works and the way to bypass it. This is done in chapter 9.

```
scanr :: (a → b → b) → b → [a] → [b]
scanr _ v [] = [v]
scanr f v (x:xs) = x `f` (head part) : part where part = scanr f v xs

scanl :: (b → a → b) → b → [a] → [b]
scanl _ v [] = [v]
scanl f v (x:xs) = v : scanl f (v `f` x) xs
```

x Scan as a fold

We can define `scanr` using `foldr`, try to figure out a way to do so.

x Defining `scanl1` and `scanr1`

Modify these definitions and define `scanl1` and `scanr1`.

This seems like a much more convoluted recursion pattern. So why have we decided to study it? Let's see by example

x Not Quite Lisp (AOC 2015, 1)

Santa is trying to deliver presents in a large apartment building, but he can't find the right floor - the directions he got are a little confusing. He starts on the ground floor (floor 0) and then follows the instructions one character at a time.

An opening parenthesis, `(`, means he should go up one floor, and a closing parenthesis, `)`, means he should go down one floor.

The apartment building is very tall, and the basement is very deep; he will never find the top or bottom floors.

For example:

- `((()))` and `()()()` both result in floor 0.
- `((((and ((()((both result in floor 3.`
- `)()(((` also results in floor 3.
- `(())` and `))()` both result in floor -1 (the first basement level).
- `)))` and `))()())` both result in floor -3.

Write a function `parse :: String → Int` which takes the list of parenthesis as a string in input and gives the correct integer as output.

This is quite simple using folds.

```
parse :: String → Int
parse = foldl (\x y → if y == '(' then x+1 else x - 1) 0
```

But every AOC question always has a part 2!

x Not Quite Lisp, 2

Now, given the same instructions, find the position of the first character that causes him to enter the basement (floor -1). The first character in the instructions has position 1, the second character has position 2, and so on.

For example:

- `)` causes him to enter the basement at character position 1.
- `((()))` causes him to enter the basement at character position 5.

Make a function `ans` which takes the list of parenthesis as a string in input and output the position (1 indexed) of the first character that causes Santa to enter the basement

If we had no idea of scans, this would be harder. In this case, it is just a simple replacement.

```
ans :: String → Int
ans = length.takeWhile (≠ -1).scanl (\x y → if y == '(' then x+1 else x-1) 0
```

The `takeWhile` chooses all the floors we reach before -1 . As 0th floor is counted (as it is the scan on empty list), we will have a list of `length` as much as the position of the character that caused us to enter -1 .

Now, here is a coincidence we didn't expect. We were told that AOC 2015's first question was a good `foldl` to `scanl` example. What I was not prepared to see was scanning showing up in AOC 2015's third question as well.

x Perfectly Spherical Houses in a Vacuum (AOC 2015)

Santa is delivering presents to an infinite two-dimensional grid of houses.

He begins by delivering a present to the house at his starting location, and then an elf at the North Pole calls him via radio and tells him where to move next. Moves are always exactly one house to the north (^), south (v), east (>), or west (<). After each move, he delivers another present to the house at his new location.

However, the elf back at the north pole has had a little too much eggnog, and so his directions are a little off, and Santa ends up visiting some houses more than once. How many houses receive at least one present?

For example:

- `>` delivers presents to 2 houses: one at the starting location, and one to the east.
- `>v<` delivers presents to 4 houses in a square, including twice to the house at his starting/ending location.
- `^v^v^v^v^v` delivers a bunch of presents to some very lucky children at only 2 houses.

Create function `solve1 :: String → Int` which takes the list of instructions as string in input and outputs the number of houses visited.

x Perfectly Spherical Houses in a Vacuum II

The next year, to speed up the process, Santa creates a robot version of himself, Robo-Santa, to deliver presents with him.

Santa and Robo-Santa start at the same location (delivering two presents to the same starting house), then take turns moving based on instructions from the elf, who is eggnoggedly reading from the same script as the previous year.

This year, how many houses receive at least one present?

For example:

- `^v` delivers presents to 3 houses, because Santa goes north, and then Robo-Santa goes south.
- `^>v<` now delivers presents to 3 houses, and Santa and Robo-Santa end up back where they started.
- `^v^v^v^v^v` now delivers presents to 11 houses, with Santa going one direction and Robo-Santa going the other.

Create function `solve2 :: String → Int` which takes the list of instructions as string in input and outputs the number of houses visited.

We will also briefly talk about something called Segmented Scan.

≡ Segmented Scan

A scan can be broken into segments with flags so that the scan starts again at each segment boundary. Each of these scans takes two vectors of values: a data list and a flag list. The segmented scan operations present a convenient way to execute a scan independently over many sets of values.

For example, a segmented looks like is:

```
1 2 3 4 5 6 -- Input
T F F T F T -- Flag
1 3 6 4 9 6 -- Result
```

We will name this function `segScan :: (a → a → b) → [Bool] → [a] → [b]`.

The implementation of function is as follows

```
segScan :: (a → a → b) → [Bool] → [a] → [b]
segScan f flag str = scanl (\r (x,y) → if x then y else r `f` y) (head str) (tail (zip flag str))
```

This might seem complex but we are merely `zip`-ing the flags and input values, and defining a new function, say `g` which applies the function `f`, but resets to `y` (the new value) whenever `x` (the flag) is `True`. The `head` and `tail` are to ensure that the first element is the beginning of the first segment.

This will be the end of my discussion of this. The major use of segmented scan is in parallel computation algorithms. A rather complex quick sort parallel algorithm can be created using this as the base.

§7.1.4. Exercises

}

Introduction to Datatypes

Shubh Sharma

§8.1. Datatypes (Once Again)

In Chapter 4 we saw how Haskell datatypes correspond to sets of values. Like `Integer` is the set of all integers and `String → Bool` is the set of all functions that take in a `String` as an argument and return a `Boolean` as their output. This was the first time we gave explicit attention to datatypes and learned the following:

≡ Types 1

A **Datatype**, in its simplest form, is the name of a set.

In Section §6.1., where we defined polymorphic functions, the *shape* and *behaviour* of an element were 2 properties that we built off of.

As a small recap, consider function `elem`, this is a function which checks if a given element belongs to a given list. The input requires to be a list of elements of a type, such that there is a notion of equality between types.

```
elem :: Eq a => a -> [a] -> Bool
elem _ []      = False
elem e (x : xs) = e == x || elem e xs
```

Our requirements for the function are very clearly mentioned in the type. We are starting with a type `a` which has a notion of equality defined on it, as depicted by `Eq a`, and our arguments are an element of the type `a` and a list of elements of the type, that is, `[a]`. Here we used datatypes to specify the properties of the elements that we use. So we extend the previous definition.

≡ Types 2

A **Datatype** is the name of a *homogenous* collection of object, where the common properties, like the shape of elements, is depicted in the name.

Some examples of datatypes we have already seen are:

- `[Integer]`, which is the collection of lists of integers.
- `Maybe Char`, which is the collection of characters along with the extra element `Nothing`.
- `Integer → String`, which is the collection of functions with their domain as the set of integers and range as the set of strings.

This definition suggests that datatypes can be used to *structure* the data we want to work with. And this is actually something we have seen before!

In Section §4.1., we saw operations on sets such as

- `(A, B)` being analogous to \times **cartesian product**.
- `Either A B` being analogous to $\dot{\cup}$ **disjoint union**.

Here we will spend some time to see how we can define datatypes like these on our own.

Before getting to defining our own datatypes, it's good to remember what the purpose of datatypes is: The point of datatype is to make thinking about programs simpler, for both the programmer and Haskell. This is done in the following ways:

- Types indicate the *shape* of elements and can add information about the functions, for example:
 - `Either [Integer] Bool` tells us that every element of the type is either a list of integers, or a boolean value.
 - `Eq a => a -> [a] -> Maybe Integer` tells us that `a` has a notion of equality defined on it, and the output should be an integer, but the function can potentially fail (that is return `Nothing`).
- Types tell the compiler information about domain and codomain of functions, which makes it possible for Haskell to prove that a huge class of functions is complete, that is, it returns a well defined answer on all inputs.

We will now see how to define our own types.

§8.2. Type Synonyms

The simplest way in which we can define our own types is by giving another name to an already existing type. This is done using the keyword `type` as follows:

type aliases

```
type Point = (Integer, Integer)
type String = [Char] -- This is how Haskell defines String!

type Name = String
type Age = Integer

type Person = (Name, Age)
```

note that any type defined using the keyword `type` is simply an alias for another type and Haskell does not treat it any differently.

Nonetheless, this can be very helpful for interpreting the type for a human. For example the type `Person` which is an alias for `(String, Integer)`, when written as `(Name, Age)`, is very clearly meant to be a pair containing the name of a person, and their age.

§8.3. Finite Types

The step, which is really a big one, is that we will now define our own types, which contain the values that we create, this is done using the `data` keyword.

A finite types

```
data Colour = Red | Green | Blue

data Bool = True | False      -- This is how Haskell defines Bool!
data Ordering = LT | EQ | GT  -- This is also a type defined by Haskell

data Coin = Heads | Tails
```

The last example is there to emphasize that the `data` keyword really creates new types. The `Coin` type is a 2-element type, but is not the same as `Bool` and Haskell will give a type error if its used in its place. Each element of a type defined like this is called a **constructor**, which is name that will get its justification by the end of the chapter.

To define a function out of a finite type, one needs to define the output all all constructors, for example:

```
isRed :: Colour → Bool
isRed Red    = True
isRed Blue   = False
isRed Green  = False
```

Defining finite types is really helpful when one wants to have a finite number of variants in a type, for example, there are a finite number of chess pieces, in languages that do not have a syntax that lets us do something like this, one would make do with strings. The benefit of these finite types is that now Haskell will make sure that the functions are only defined on intended values (unlike all possible strings), and will also give warnings if any function is not defined on all variants.

X Finite Types

Define the types `Month`, `Day` of the week and `DiceHead` as finite types.

§8.4. Product Types

These are what we get when take ⊗ **cartesian product** of other, simpler types. The purpose of product types is to define data, that has multiple smaller components. For example:

- A `Point` on a 2D grid which has 2 integer components.
- A `Profile` representing a profile on a dating app, which would contain the person's name, their age, some images and more information about them. We will be using the first example to keep things simple.
- Complex Numbers can be thought of as having 2 components, real and imaginary.

The first way to create a product, which is something we have already seen before is a tuple.

```
type Point = ( Int , Int )
            --   X-coord Y-coord
```

And we can extract components using `fst` and `snd` functions. (Note `Point` is just a synonym for `(Int, Int)`).

Another way to do so is to use the `data` keyword again in a much more powerful way!

```
data Point = Coord Int Int -- Constructors can take inputs!
           --   X   Y

x_coord, y_coord :: Point → Int
x_coord (Point x _) = x
y_coord (Point _ y) = y
-- we use underscores to state that we don't care about the value
```

Here we need to define our own functions to extract components as `Point` is different from `(Int, Int)`.

The second important thing to highlight here is that constructors are functions! They are called so because they “construct” an element of the type associated with them, like `Coord` constructs elements of type `Point`, in fact Haskell will even give us a type for it. Constructors for finite types can be thought of as functions that take 0 arguments (so, they just behave as values).

```
>>> :t Coord
Coord :: Int → Int → Point
```

Since defining a product type, and then defining functions to extract the components is a fairly common practice, Haskell has another way to define product types.

```
data Point = Coord {
  x_coord :: Int,
  y_coord :: Int
}

-- This is how one can create an element!
origin :: Point
origin = Coord { x_coord = 0, y_coord = 0 }
```

These are called **Records** it's a syntactic sugar, which means internally Haskell treats it just like the previous way of defining product types, so `Coord 0 0` also works. But now we have the 2 functions `x_coord` and `y_coord` defined!

Dating Profile

As described above, the profile of a dating app can be also thought of as a product type, one which is more complicated than a simple point in the 2d grid. Define the type `Profile` and try to see how elaborate you can make it. A fun rabbit hole to dive into would be to see how dating apps work.

Complex Numbers

Define the datatype `Complex`, we will be looking at this again in later sections of the chapter.

§8.5. Parametric Types

We will once again extend the use of `data` keyword using ideas from Section §6.1.

We compared product types with tuples, we even treated `Point` as a special case `(Int, Int)` for a while. Turns out we can define our tuples, in its full generality as follows:

```
Tuple A B = Pair A B

ex :: Tuple Int String
ex = Pair 5 "Heyy!"
```

Here `Tuple` is called a **parametric type**, and this is similar to how haskell defines its tuples, it just adds an extra syntactic sugar so we can write it as `(a,b)`.

Some other **parametric types** that we have seen before, and we will be discussing in depth in the next section are:

- `Maybe a`
- `Either a b`
- `[a]`, The list type
- The function type `a → b`

§8.6. Sum Types

Sum types are what type theory people like to call  **disjoint union**. And already have seen everything we need to construct sum types:

- Finite Types
- Viewing constructors as functions
- Parametric Types

The purpose of having sum types is to have a collection of many possible *variants* in a type. This is similar to what we did with **Finite types** but we can have an entire collection as a variant with the help of **Parametric types**. Here are some examples:

```
IntOrString = I Integer | S String

Maybe a = Just a | Nothing    -- This is how Haskell defines Maybe types!
Either a b = Left a | Right b -- This is how Haskell defines Either types!

Shape = Circle { radius :: Double } -- Record syntax works!
      | Square { side :: Double }
      | Rectangle { len :: Double, width :: Double }
-- Remember, the records are just syntactic sugar, which are converted to
-- Shape = Circle Double | Square Double | Rectangle Double Double
```

Just like finite types, to define a function on a sum type, one needs to define it on all variants. This is also called Pattern Matching!

```
area :: Shape → Double
area (Circle r)      = pi * r * r
area (Square s)      = s * s
area (Rectangle l b) = l * b
-- If this doesn't make a lot of sense, read the comment below the
-- definition of the type.
```


Better Dating Profile

Think of some interesting questions and possible answers for those questions / information bits for a dating profile and incorporate it into you `Profile` type.

§8.7. Inductive Types

Inductive types will be the final tool in the type construction toolbox that we will be looking at. While it can be considered as a slight modification of the previous methods of building types, we will give it some special attention.

§8.7.1. Inductive Types (as a Mathematician)

We defined a  **set** as a well-defined collection of objects. So consider the following description:

$$B = \{\text{Set of squares reachable by a bishop} \\ | \text{ given that the bottom left square is in the set}\}$$

Here is a quick refresher on the relevant rules of chess:

- There is an 8x8 square grid and each piece lies inside a square.
- Bishop is one of the chess pieces that can only move diagonally in a line, but it is allowed to move as far as possible.

One can now create the set B by starting at the bottom left square and one by one adding squares, where the bishop can reach, to set. Here we say that the set B was generated by the bottom left under the bishop movement rules. Here, there was one other important piece of information other than the “base case” and the “operation”, the extra structure imposed by the chess board itself, specifically, the operations $\langle \text{go top right} \rangle \rightarrow \langle \text{go top left} \rangle$ is the same as $\langle \text{go top left} \rangle \rightarrow \langle \text{go top right} \rangle$ and the board is restricted to an 8×8 grid.

If that was not the case then we would call the set **freely generated**. In a freely generated set, a sequence of operations uniquely maps to an element of the set. In such cases we give the description of the set by simply giving the element and the operations. Consider the following description of natural numbers as an example:

§8.7.1.1. Natural Numbers as Inductive Types

Let \mathbb{N} be the set freely generated by the following:

- The element $0 :: \mathbb{N}$
- The operation $\text{succ} :: \mathbb{N} \rightarrow \mathbb{N}$

Here the names are suggestive, but if simply follow the rules for freely generated sets, we get the following:

- We know that 0 is in the set.
- That means $\text{succ } 0$ is in the set.
- Which means $\text{succ } (\text{succ } 0)$ is in the set.
- and so on...

So we will end up with the set

$$\{ 0, \text{succ } 0, \text{succ succ } 0, \text{succ succ succ } 0, \text{succ succ succ succ } 0 \dots \}$$

Now we can

This way is very similar to how mathematicians usually formalize natural numbers.⁶

⁶The usual way to define natural numbers was written by Giuseppe Peano and are called the ‘Peano Axioms’ which involve a bunch of rules, whose relevant part can be summarized as :

- 0 is a natural number
- Natural numbers are closed under the succ operation
- For each number x that is not 0, there is a unique number y such that $x = \text{succ } y$
- 0 is not the successor of any number

These “freely generated sets” are what programmers call Inductive types, and one can define the type of natural numbers in haskell as follows:

```
λ nat
data Nat = Z | Succ Nat

three :: Nat
three = Succ (Succ (Succ Z))
```

And functions on a natural number would be usually given by a recursive function:

```
add' :: Nat → Nat → Nat
add' Z n = n
add' (Succ m) n = Succ (add' m n)
```

We can also define functions to convert between Integers and Natural Numbers:

```
λ nat and integer
natToInteger :: Nat → Integer
natToInteger Z = 0
natToInteger (Succ n) = natToInteger n + 1

integerToNat :: Integer → Maybe Nat
integerToNat n | n < 0 = Nothing
               | n == 0 = Z
               | n > 1 = Succ (integerToNat (n-1))
```

✕ Exercise

Natural numbers are the default way people count things. A lot of the haskell functions that involve counting, like `(!!)`, `takeWhile`, `drop` and so on are functions that can potentially fail because of an attempt to access a negative index. redefine these functions our definition of natural numbers (`λ nat`).

✕ Functions on naturals

Define versions of functions `max`, `sum`, `prod` (product), `min` and `=` for natural numbers. Note that you would have to use new names.

§8.7.1.2. Lists as Inductive Types

Another interesting example of an inductive type is the set of list of elements of type `A`.

Given a set/type `A` we can define the set of list elements using the following:

- `[] :: [A]`, the empty list.
- For each element `a :: A`, a function `(a :) :: [A] → [A]`.

We leave it to the reader to show that this inductive type generates the set of all lists of elements of type `A`. We will justify for the example `[0, 1, 2, 3]`

- `[]` belongs to the set \mathbb{Z}
- Then `3 : []` belongs to the set \mathbb{Z}
- Then `2 : 3 : []` belongs to the set \mathbb{Z}
- Then `1 : 2 : 3 : []` belongs to the set \mathbb{Z}

The first 2 rule are given in the definition, the next 2 are subsumed in the definition of “free generation”.

- Then $0 : 1 : 2 : 3 : \square$ belongs to the set \mathbb{Z}

And we treat $0 : 1 : 2 : 3 : \square$ as $[0, 1, 2, 3]$.

In Haskell, we cannot write infinitely many constructors in the definition of a type, so we instead define it as follows:

```
λ list
data List A = Nil | Cons A (List A)
```

Haskell will not let us use `[]`, this is syntactic sugar given by the compiler a user (like us) cannot give our own definitions to, so we will use `Nil` and `Cons` instead.

Fixing as `Integer` for now `Nil` represents `[]` and `Cons` takes an integer `n` and gives the constructor `n:`. This is our workaround for having multiple constructors. The above definition (apart from the syntactic sugar) is how Haskell internally defines lists.

This idea is very much inspired by the concept of [Currying](#) which was discussed in Section §6.2.1..

§8.7.2. Inductive Types (as a Programmer)

As a programmer, inductive types are used to indicate that the elements of a type are created from other smaller elements of the type. This is pretty much the same as what the mathematician thinks, as it should be, but a programmer puts more emphasis on the fact that the type is the *blueprint* of the elements in it, that is, the *shape* of the elements is reflected in the type.

To understand this better, consider the example where you want to write a calculator. A calculator takes a simple arithmetic expression and evaluates it.

§8.7.2.1. Calculator

For our purposes, we say that our calculator can compute:

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

The plan will be to have an inductive type `Expr` of expressions (because tiny expressions combine to give big expressions), which we define as follows:

```
λ expression
data Expr = Val Double
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Exp Expr Expr
          | Neg Expr
```

Here the goal of the type was to specify the structure of the data (arithmetic expression) we want to be working with, let's see a few examples!

The expression $3 + 5 * 10 + \frac{8^3}{2}$ corresponds to

λ **expr example**

```
ex :: Expr
ex = Add (Val 3.0)
      (Add (Mul (Val 5.0)
                (Val 10.0))
          (Div (Exp (Val 8.0)
                    (Val 3.0))
              (Val 2.0)))
```

✕ Evaluate and extend

Write a function `eval :: Expr → Double` that takes an expression and returns its value. The potential failure case here is division by 0. To deal with it, either add an failure value to the expression type, or make the function have a `Maybe` output.

Also try extending the Expression type to include more operations.

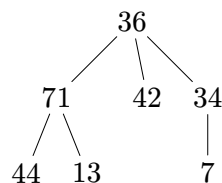
§8.7.2.2. Trees as Inductive Types

For those with keen eyes and good memory the shape of `ex` should remind you of the discussion in *Why Trees?* section in Section §1.8..

On the topic of trees, while working with such inductive one finds that all inductive datatypes follow a tree structure, this is a result of *free generation*. Trees happen to be a ubiquitous data-structure (way to structure data) in computer science and has applications everywhere. The following is a very tiny subset of those:

- Compilers (like both haskell and our calculator)
- File Systems
- Databases
- Data representation formats like JSON and XML
- Data Compression (huffman encoding)
- Space partitioning (oct-trees and quad-trees)

So we now define trees, recall \equiv **tree**, which defines a tree as a meaningful structure on data involving a main **root** node, and each node having 0 or more children, as shown in the following diagram.



Looking at the structure, we can define a tree as follows in haskell:

λ **tree**

```
data Tree a = Node { value :: a, children :: [Tree a] }
```

And the above tree can be represented as follows:

```
ex :: Tree Integer
ex = Node 36
    [ Node 71 [Node 44 [], Node 13 []]
    , Node 42 []
    , Node 37 [Node 7 []]]
```

x Tree Functions

Define the following functions for the tree datatype:

- `depth :: Tree a → Nat`, this defines the longest path one can take starting from the route, for example, the `depth ex` is 2.
- `size :: Tree a → Nat`, this defines the number of nodes in a tree, for example `size ex` is 7.

Also define versions of the `elem` and `sum` functions for the `Tree` datatype.

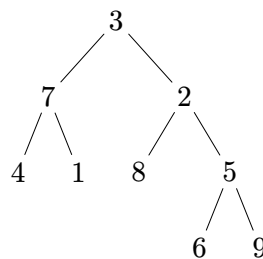
§8.7.2.3. Binary Trees

Binary Trees are a special case of trees, where each node has either exactly 2, or 0 children. Nodes with 0 children are called **leaves**.

Out of the uses cases mentioned for trees the following involve binary trees:

- Compilers (for functional languages)
- Internals of File Systems (BTRFS)
- Databases
- Data compression (Huffman Encoding)

The following is an example of a binary tree:



The following is how our definition of a Binary Tree can be written in haskell:

λ Btree

```
data BTree a = BNode {left :: Btree a, Val :: a, right :: Btree a}
               | Leaf a
```

and the example above can be written as:

λ Btree ex

```
bex :: Btree Integer
bex = BNode (BNode (Leaf 4) 7 (Leaf 1))
           3
           (BNode (Leaf 8) 2 (BNode (Leaf 5
                                   5
                                   (Leaf 9))))
```

x Binary Tree Functions

Define all of the functions of the x Tree Functions for binary trees.

We will see how these datatypes are used in Section §11.1..

Computation as Reduction

Shubh Sharma

§9.1. computation (feel free to change it)

Complexity

Arjun Maneesh Agarwal

§10.1. complexity (feel free to change it)

Advanced Data Structures

Arjun Maneesh Agarwal

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

§11.1.1. Stacks and Queues

Type Classes

Ryan Hota

§12.1. **typeclasses** (feel free to change it)

Monads

Ryan Hota

§13.1. Monad (feel free to change it)