

# 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](mailto:feedback_host@mailthing.com)

Last updated August 09, 2025.

*To someone*

# Table of Contents

Table of Contents	iii
-------------------	-----

Basic Theory	1
--------------	---

§1.1. Precise Communication	1
-----------------------------	---

§1.2. The Building Blocks	1
---------------------------	---

§1.3. Values	1
--------------	---

⊸ mathematical value	1
----------------------	---

§1.4. Variables	2
-----------------	---

⊸ mathematical variable	2
-------------------------	---

§1.5. Well-Formed Expressions	3
-------------------------------	---

⊸ checking whether mathematical expression is well-formed	3
--	---

⊸ well-formed mathematical expression	3
---------------------------------------	---

§1.6. Function Definitions	4
----------------------------	---

§1.6.1. Using Expressions	4
---------------------------	---

§1.6.2. Some Conveniences	5
---------------------------	---

§1.6.2.1. Where, Let	5
----------------------	---

§1.6.2.2. Anonymous Functions	5
-------------------------------	---

§1.6.2.3. Piecewise Functions	6
-------------------------------	---

§1.6.2.4. Pattern Matching	6
----------------------------	---

§1.6.3. Recursion	7
-------------------	---

§1.6.3.1. Termination	7
-----------------------	---

⊸ termination of recursive definition	8
---------------------------------------	---

§1.6.3.2. Induction	8
---------------------	---

⊸ principle of mathematical induction	8
---------------------------------------	---

§1.6.3.3. Proving Termination using Induction	10
---	----

§1.7. Infix Binary Operators	10
------------------------------	----

⊸ infix binary operator	11
-------------------------	----

§1.8. Trees	11
-------------	----

§1.8.1. Examples of Trees	11
---------------------------	----

§1.8.2. Making Larger Trees from Smaller Trees	11
--	----

§1.8.3. Formal Definition of Trees	12
------------------------------------	----

⊖ checking whether object is tree . . . . .	13
⊖ tree . . . . .	13
§1.8.4. Structural Induction . . . . .	14
⊖ structural induction for trees . . . . .	14
§1.8.5. Structural Recursion . . . . .	14
⊖ tree size . . . . .	14
§1.8.6. Termination . . . . .	14
⊖ tree depth . . . . .	15
§1.9. Why Trees? . . . . .	15
§1.9.1. The Problem . . . . .	16
§1.9.2. The Solution . . . . .	16
⊖ abstract syntax tree . . . . .	17
§1.9.3. Exercises . . . . .	17
 Installing Haskell . . . . .	 23
§2.1. Installation . . . . .	23
§2.1.1. General Instructions . . . . .	23
§2.1.2. Choose your Operating System . . . . .	23
§2.1.2.1. Linux . . . . .	23
§2.1.2.2. MacOS . . . . .	24
§2.1.2.3. Windows . . . . .	25
§2.2. Running Haskell . . . . .	25
§2.3. Fixing Errors . . . . .	26
§2.4. Autocomplete . . . . .	26
 Basic Syntax . . . . .	 27
§3.1. The Building Blocks . . . . .	27
§3.2. Values . . . . .	27
⊖ value . . . . .	27
§3.3. Variables . . . . .	27
⊖ variable . . . . .	27
λ double . . . . .	27
§3.4. Types . . . . .	28
§3.4.1. Using GHCi to get Types . . . . .	28
λ :type +d . . . . .	28

λ :type .....	29
§3.4.2. Types of Functions .....	29
λ functions with many inputs .....	29
§3.5. Well-Formed Expressions .....	29
≡ checking whether expression is well-formed .....	30
≡ well-formed expression .....	30
§3.6. Infix Binary Operators .....	31
λ using infix operator as function .....	31
λ using function as infix operator .....	32
§3.6.1. Precedence .....	32
≡ left-associative .....	32
≡ right-associative .....	33
§3.7. Logic .....	33
§3.7.1. Truth .....	33
§3.7.2. Statements .....	33
λ simplest logical statements .....	34
λ type of < .....	34
§3.8. Conditions .....	34
λ condition on a variable .....	34
§3.8.1. Logical Operators .....	35
≡ logical operator .....	35
λ not .....	36
§3.8.1.1. Exclusive OR aka XOR .....	36
≡ XOR .....	36
§3.9. Function Definitions .....	36
§3.9.1. Using Expressions .....	36
λ basic function definition .....	37
λ function definition with explicit type .....	37
λ xor .....	37
§3.9.2. Some Conveniences .....	37
§3.9.2.1. Piecewise Functions .....	37
λ guards .....	38
λ basic usage of guards .....	38
λ guards .....	39
λ otherwise .....	39
λ if-then-else .....	39

λ if-then-else example .....	39
§3.9.2.2. Pattern Matching .....	39
λ exhaustive pattern matching .....	40
λ pattern matching .....	40
λ unused variables in pattern match .....	40
λ wildcard .....	40
λ using other functions in RHS .....	40
λ pattern matches mixed with guards .....	40
λ trivial case .....	41
λ non-trivial case .....	41
§3.9.2.3. Where, Let .....	41
λ where .....	41
λ let .....	41
§3.9.2.4. Without Inputs .....	41
λ function definition without input variables ..	42
§3.9.2.5. Anonymous Functions .....	42
λ basic anonymous function .....	42
λ multi-input anonymous function .....	42
§3.9.3. Recursion .....	42
λ factorial .....	43
λ binomial .....	43
λ naive fibonacci definition .....	43
§3.10. Optimization .....	43
λ computation of naive fibonacci .....	43
λ fibonacci by tail recursion .....	43
λ computation of tail recursion fibonacci .....	44
§3.11. Numerical Functions .....	44
≡ Integer and Int .....	44
≡ Rational .....	44
≡ Double .....	45
≡ Integer and Int .....	45
≡ Rational .....	45
≡ Double .....	45
λ Implementation of abs function .....	46
§3.11.1. Division, A Trilogy .....	46
λ A division algorithm on positive integers by repeated subtraction .....	49

§3.11.2. Exponentiation .....	49
λ A naive integer exponentiation algorithm . . .	51
λ A better exponentiation algorithm using divide and conquer	51
§3.11.3. gcd and lcm .....	52
λ Fast GCD and LCM .....	52
§3.11.4. Dealing with Characters .....	53
§3.12. Mathematical Functions .....	53
§3.12.1. Binary Search .....	55
≡ Hi-Lo game .....	55
λ Square root by binary search .....	58
§3.12.2. Taylor Series .....	58
λ Log defined using Taylor Approximation . . .	58
λ Sin and Cos using Taylor Approximation . . .	59
§3.13. Exercises .....	59
≡ Newton–Raphson method .....	60



# Basic Theory

## §1.1. Precise Communication

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

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

When writing any form of mathematical expression, it is the expectation that it is meant to be read by humans, and convince them of some mathematical proposition.

On the other hand, haskell code is not *primarily* meant to be read by humans, but rather by machines. The computer reads haskell code, and interprets it into steps for manipulating some expression, or doing some action.

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

However, in the context of Haskell, computers, being machines, are extremely unimaginative, and do not possess any inherent pattern-sensing abilities. Unless we spell out the details for them in excruciating detail, they are not going to understand what we want them to do.

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

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

## §1.2. The Building Blocks

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

- Symbols: such as  $0, 1, 2, 3, x, y, z, n, \alpha, \gamma, \delta, \mathbb{N}, \mathbb{Q}, \mathbb{R}, \in, <, >, f, g, h, \Rightarrow, \forall, \exists$  etc. Along with;
  - Expressions: which are sentences or phrases made by chaining together these symbols, such as
    - $x^3 \cdot x^5 + x^2 + 1$
    - $f(g(x, y), f(a, h(v), c), h(h(h(n))))$
    - $\forall \alpha \in \mathbb{R} \exists L \in \mathbb{R} \forall \varepsilon > 0 \exists \delta > 0 \mid x - \alpha \mid < \delta \Rightarrow \mid f(x) - f(\alpha) \mid < \varepsilon$
- 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  $\pi$
- The order  $<$  on  $\mathbb{N}$
- The function of squaring a real number :  $\mathbb{R} \rightarrow \mathbb{R}$

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

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

In fact, as we see in the last example, we don't even currently know the exact value of  $d$ .

The famous "Twin Primes Conjecture" is just about whether  $d == 2$  or not.

So, the moral of the story is that even if we don't know what the exact value is,

we can still know that it is some  $\div$  **mathematical value**,

as it does not change from scenario to scenario and remains constant, even though it is an unknown constant.

## §1.4. Variables

$\div$  **mathematical variable**

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

The following examples should clarify further.

For example, consider the following function definition -

$$\begin{aligned} f : \mathbb{R} &\rightarrow \mathbb{R} \\ f(x) &:= 3x + x^2 \end{aligned}$$

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

Consider the following theorem -

**Theorem** Adding 1 to a natural number makes it bigger.

**Proof** Take  $n$  to be an arbitrary natural number.

We know that  $1 > 0$ .

Adding  $n$  to both sides of the preceding inequality yields

$$n + 1 > n$$

Hence Proved !! ■

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

Here is another theorem -

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

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

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

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

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

Hence Proved! ■

Here,  $f$  is a  $\Rightarrow$  **mathematical variable** as it isn't any one specific  $\Rightarrow$  **mathematical value**, but rather represents an arbitrary element from the set of all  $\mathbb{N} \rightarrow \mathbb{N}$  strictly increasing functions. It has been used to show a certain fact that holds for any natural number.

## §1.5. Well-Formed Expressions

Consider the expression -

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

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.

### $\Rightarrow$ checking whether mathematical expression is well-formed

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

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

The procedure is as follows -

Given an expression  $e$ ,

- first check whether  $e$  is
  - a  $\Rightarrow$  **mathematical value**, or
  - a  $\Rightarrow$  **mathematical variable**
 in which cases  $e$  passes the check and is a **well-formed expression**.

Failing that,

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

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

### $\Rightarrow$ well-formed mathematical expression

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

Let us use  $\Rightarrow$  **checking whether mathematical expression is well-formed** to check if  $x^3 \cdot x^5 + x^2 + 1$  is a well-formed expression.

( We will skip the check of whether something is a valid input or not, as that notion is still not very well-defined for us. )

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

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

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

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

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

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

$x$  is a well-formed expression, as it is a  $\doteq$  **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  $\doteq$  **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  $\doteq$  **mathematical variable**.

$1$  is a well-formed expression, as it is a  $\doteq$  **mathematical value**.

Done!

#### checking whether expression is well-formed

Suppose  $a, b, v, f, g$  are  $\doteq$  **mathematical values**.

Suppose  $x, y, n, h$  are  $\doteq$  **mathematical variables**.

Check whether the expression

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

is well-formed or not.

## §1.6. Function Definitions

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

Thus, it is very helpful to have a deeper understanding of how function definitions in mathematics work.

### §1.6.1. Using Expressions

In its simplest form, a function definition is made up of a left-hand side, ‘ $:=$ ’ in the middle<sup>1</sup>, and a right-hand side.

A few examples -

---

<sup>1</sup>In order to have a clear distinction between definition and equality, we use  $A := B$  to mean “ $A$  is defined to be  $B$ ”, and we use  $A = B$  to mean “ $A$  is equal to  $B$ ”.

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

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

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

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

### §1.6.2. Some Conveniences

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

#### §1.6.2.1. Where, Let

Consider the definition of the famous sine function -

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

Given an angle  $\theta$ ,

Let  $T$  be a right-angled triangle, one of whose angles is  $\theta$ .

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 the exact same thing using “where” instead of “let”.

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

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

,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 -

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

which, for convenience, can be rewritten as -

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

### §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 -

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

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

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

- $\varphi_0$  is true
- For each  $n > 0$ , if  $\varphi_{n-1}$  is true, then  $\varphi_n$  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.



**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 2-3 Color Theorem**

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

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

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

**X Square-full**

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

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

**X Same Height?**

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

### x proving the principle of induction

Prove that the following statements are equivalent -

- every nonempty subset of  $\mathbb{N}$  has a smallest element
- the  $\Leftrightarrow$  principle of mathematical induction

You can assume that  $<$  is a linear order on  $\mathbb{N}$

such that there are no elements strictly between  $n$  and  $n + 1$ .

#### §1.6.3.3. Proving Termination using Induction

So let's see the  $\Leftrightarrow$  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  $\varphi_n$  be the statement

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

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

- $\langle\langle \varphi_0 \text{ is true} \rangle\rangle$

The only natural number which is  $\leq 0$  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  $\varphi_{n-1}$  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  $\leq n-1$ ,

$\varphi_{n-1}$  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  $\varphi_n$  is proved!

Hence the theorem is proved!! ■

## §1.7. Infix Binary Operators

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

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

$$x + y$$

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

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

Such a function is called an infix binary operator<sup>2</sup>

### ⚡ infix binary operator

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

Examples include -

- + (addition)
- − (subtraction)
- × or \* (multiplication)
- / (division)

## §1.8. Trees

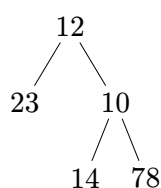
Trees are a way to structure a collection of objects.

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

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

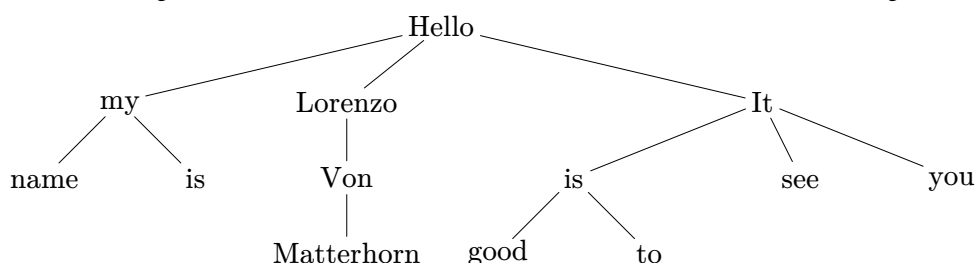
### §1.8.1. Examples of Trees

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



The line segments are what defines the structure.

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



### §1.8.2. Making Larger Trees from Smaller Trees

If we have an object -

89

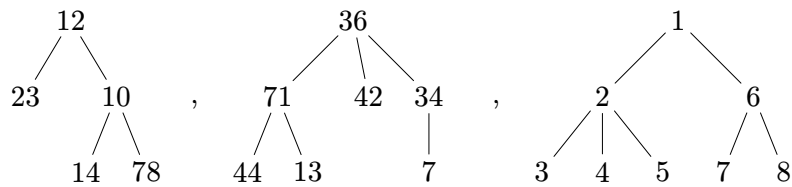
and a few trees -

<sup>2</sup>

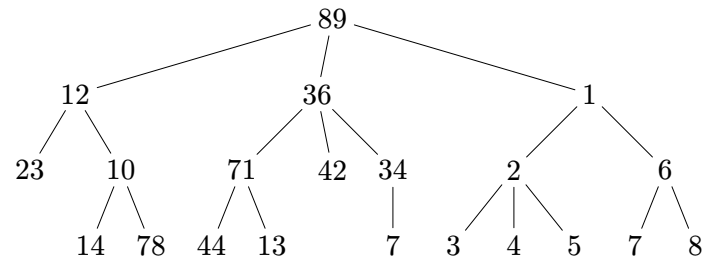
infix - because the function name is in-between the inputs

binary - because exactly 2 inputs, and binary refers to 2

operator - another way of saying function



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



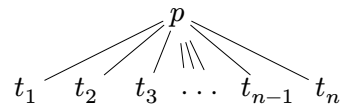
In general, if we have an object

$$p$$

and a bunch of trees

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

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



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

### §1.8.3. Formal Definition of Trees

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

The examples we've seen include trees over the set  $\mathbb{N}$ , as well as a tree over the set of English words.

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

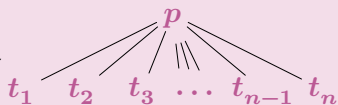
### checking whether object is tree

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

Given a mathematical object  $t$ ,

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

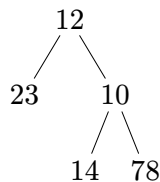
Failing that,

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

### tree

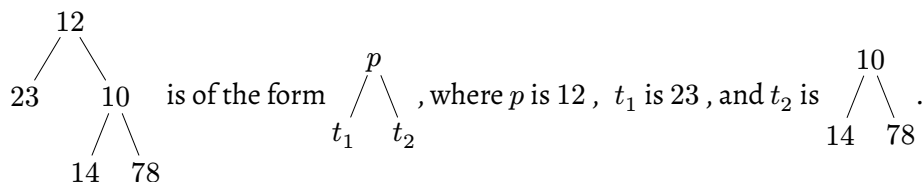
Given a set  $S$ , a **mathematical object** is said to be a **tree over  $S$**  if and only if it passes the formal checking procedure defined in  $\div$  **checking whether object is tree**.

Let us use this definition to check whether

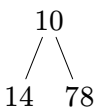


is a tree over the natural numbers.

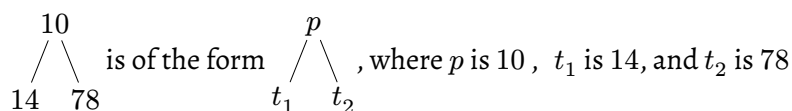
Let's start -



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.

### §1.8.4. Structural Induction

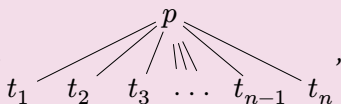
In order to prove things about trees, we have a version of the  $\equiv$  principle of mathematical induction for trees -

#### $\equiv$ structural induction for trees

Suppose for each tree  $t$  over a set  $S$ , we have a statement  $\varphi_t$ .

If we can prove the following two statements -

- For each  $s \in S$ ,  $\varphi_s$  is true

- For each tree  $T$  of the form ,

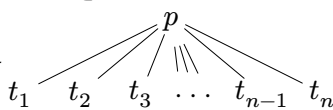
if  $\varphi_{t_1}$ ,  $\varphi_{t_2}$ ,  $\varphi_{t_3}$ , ...,  $\varphi_{t_{n-1}}$  and  $\varphi_{t_n}$  are all true,  
then  $\varphi_T$  is also true.

then  $\varphi_t$  is true for all trees  $t$  over  $S$ .

### §1.8.5. Structural Recursion

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

From the definition of  $\equiv$  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.

#### $\equiv$ tree size

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  $S$ .

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

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

### §1.8.6. Termination

Using  $\equiv$  structural induction for trees, let us prove that

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

**Proof** For each tree  $t$ , let  $\varphi_t$  be the statement

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

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

- $\langle\langle \forall s \in S, \varphi_s \text{ is true } \rangle\rangle$   
 $\text{size}(s) := 1$ , so the definition terminates immediately.
- $\langle\langle \text{For each tree } T \text{ of the form } \dots \text{ then } \varphi_T \text{ is also true} \rangle\rangle$   
 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 “ $\text{size}(T)$ ”,  
 being defined as an addition of these terms,  
 must also terminate.  
 (since addition of finitely many terms always terminates)

Hence  $\varphi_T$  is proved!

Hence the theorem is proved!! ■

### x tree depth

Fix a set  $S$ .

$\div$  **tree depth**

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

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

$$\text{depth} \left( \begin{array}{c} p \\ \swarrow \quad \downarrow \quad \searrow \\ t_1 \quad t_2 \quad t_3 \quad \dots \quad t_{n-1} \quad t_n \end{array} \right) := 1 + \max_{1 \leq i \leq n} \{\text{depth}(t_i)\}$$

1. Prove that the definition of the function “depth” terminates on any tree over  $S$ .
2. Prove that for any tree  $t$  over the set  $S$ ,  

$$\text{depth}(t) \leq \text{size}(t)$$
3. When is  $\text{depth}(t) == \text{size}(t)$  ?

### x Exercise

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

Using the  $\div$  **principle of mathematical induction**,  
 prove  $\div$  **structural induction for trees**.

## §1.9. Why Trees?

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

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

### §1.9.1. The Problem

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

How can we do this?

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

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

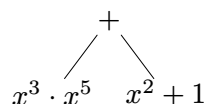
So, for dumb Haskell to do this (figure out the values of expressions, which is quite an important ability), a vital requirement is to be able to easily put expressions in the form  $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$ .

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.

### §1.9.2. The Solution

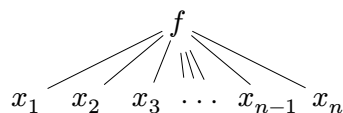
One way to make this easier to represent the expression in the form of a tree -

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

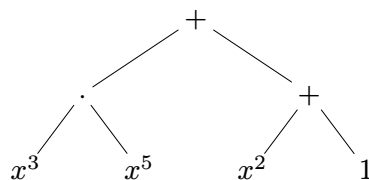


, it becomes obvious what the function is and what the inputs are to which it is applied.

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

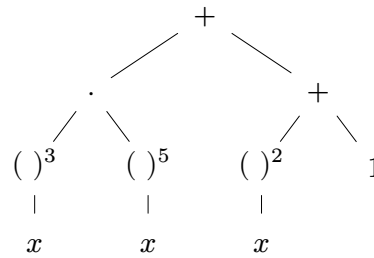


But why stop there, we can represent the sub-expressions ( such as  $x^3 \cdot x^5$  and  $x^2 + 1$  ) as trees too -



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

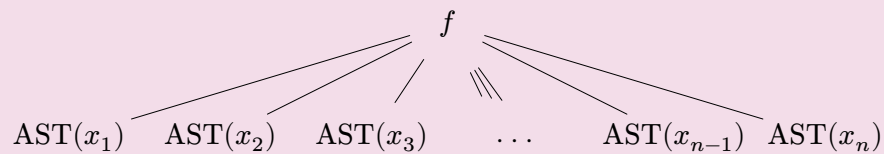
The **abstract syntax tree of a well-formed expression** is defined by applying the “function” **AST** to the expression.

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

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



### §1.9.3. Exercises

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

#### x Turbo The Snail(IMO 2024, P5)

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

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

<sup>3</sup>Atleast one author is of the opinion:

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

### [x Points in Triangle](#)

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

### [x Joining Points\(IOI 2006, 6\)](#)

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

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

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

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

Hint: Try using the ‘trick’ you discovered in [x Points in Triangle](#).

### [x Usmons\(USA TST 2015, simplified\)](#)

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

### [x Battery](#)

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

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

### [x Seven Tries \(Russia 2000\)](#)

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

Note: We know of 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 The best (trollest) codeforces question ever! (Codeforces 1028B)**

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

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

For Example

Input1 : 6 5

Output1 : 6 7

Input2 : 8 16

Output2 : 35 53

**x Rope**

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

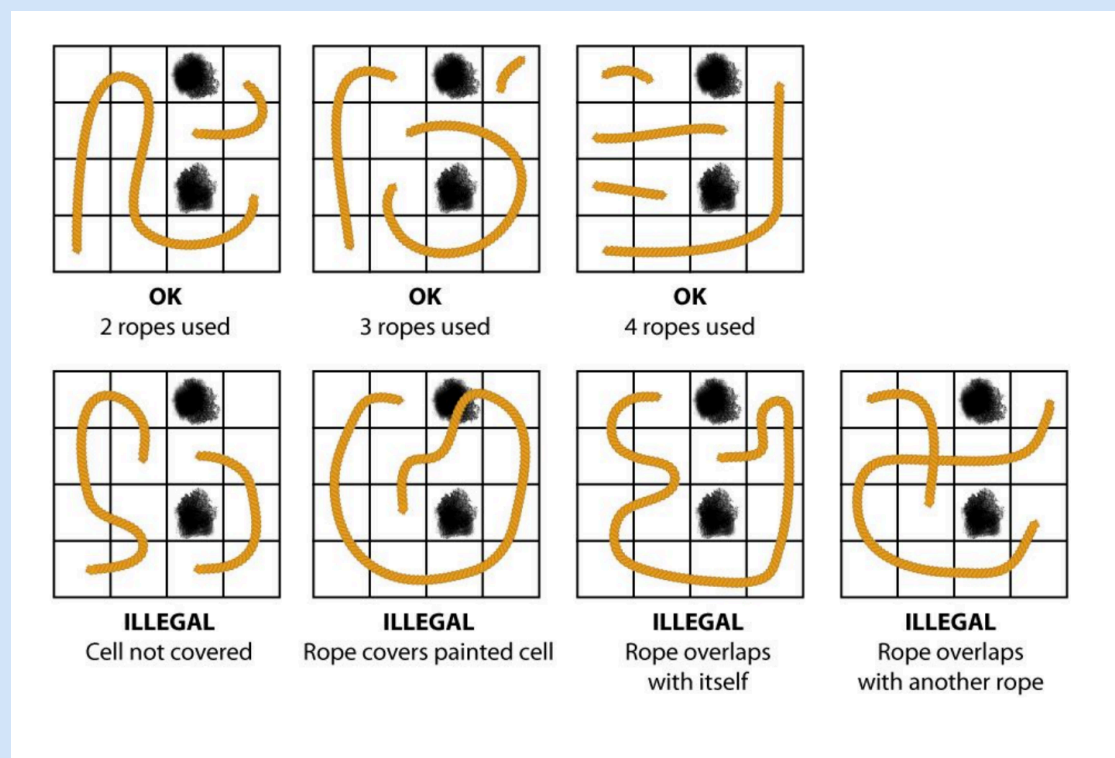


Figure out an algorithm/recipe to covering the grid using  $n + 1$  ropes leagally.

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  $5^n$**

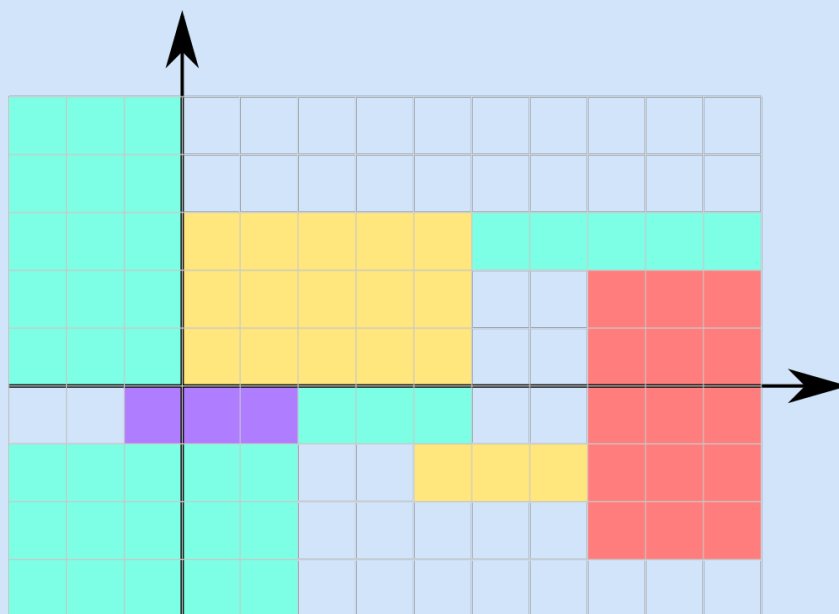
Prove that for every positive integer  $n$ , there exists an  $n$ -digit number divisible by  $5^n$ , all of whose digits are odd.

**x** This was rated 2100? (Codeforces 763B)

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

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

For example,



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

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

**x** Seating

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

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

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

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

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

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

Your task is this: Find a new seating chart (in general an algorithm/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 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

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

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

### 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 Coins (IMO 2010 P5)**

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

- (1) Choose a non-empty box  $B_j$ ,  $1 \leq j \leq 5$ , remove one coin from  $B_j$  and add two coins to  $B_{j+1}$ ;
- (2) Choose a non-empty box  $B_k$ ,  $1 \leq k \leq 4$ , remove one coin from  $B_k$  and swap the contents (maybe empty) of the boxes  $B_{k+1}$  and  $B_{k+2}$ .

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

# Installing Haskell

## § 2.1. Installation

### § 2.1.1. General Instructions

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

### § 2.1.2. Choose your Operating System

#### § 2.1.2.1. Linux

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

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

#### § 2.1.2.2. MacOS

##### 1. Install Haskell

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

##### 2. Install Visual Studio Code

Get it [here](#).

##### 3. Install HaskellSupport.

1. Read the general instructions very carefully, and ensure that you have complied with all the requirements properly.
2. Close all open windows and running processes other than wherever you are reading this.
3. Open the folder `Haskell` in Finder .
4. Open the folder `installation` in Finder.
5. Right click on the folder `MacOS` in Finder, and select `Open in Terminal`.
6. Type in `chmod +x installHaskellSupport.command` in the terminal.
7. Close the terminal window.
8. Open the folder `MacOS` in Finder.
9. Double-click on `installHaskellSupport.command`.
10. This may take a while.
11. You will know installation is complete if a new window pops up asking whether you trust authors. Click on “Trust”.



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

### § 2.1.2.3. Windows

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

## § 2.2. Running Haskell

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

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

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

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

It should reply `True` .

### § 2.3. Fixing Errors

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

To explore your options to remedy the issue, put your text cursor at the text above the squiggly line and right-click.

You have opened the QuickFix menu.

You can now choose a suitable option.

### § 2.4. Autocomplete

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

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

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

# Basic Syntax

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

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

## §3.1. The Building Blocks

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

## §3.2. Values

Haskell has values just like in math.

÷ value

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

Examples include -

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

## §3.3. Variables

Haskell also has its own variables.

÷ variable

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

The following examples should clarify further.

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

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

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

λ double

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

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

Note that `x` here is a variable.

Also, in mathematics we would write `double (x)`, but Haskell does not need those brackets.

So we can simply put some space between `double` and `x`, i.e.,

we write `double x`,

in order to indicate that `double` is the name of the function and `x` is its input.

Also, Note that the names of Haskell  $\neq$  **variables** have to begin with an lowercase English letter.

### §3.4. Types

Every  $\neq$  **value** and  $\neq$  **variable** in Haskell must have a “type”.

For example,

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

The type of an object is like a short description of the object’s “nature”.

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

#### §3.4.1. Using GHCi to get Types

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

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

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

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

`x :: T` is just Haskell’s way of saying “`x` is of type `T`”.

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

For example -

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

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

Or more generally -

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

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

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

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

### §3.4.2. Types of Functions

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

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

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

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

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

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

So we can deduce that in general,

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

and gives an output of type `T`,

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

## §3.5. Well-Formed Expressions

Of course, since we have `⊕` values and `⊕` variables, we can define “well-formed expressions” in a very similar manner to what we had before -

### ≡ checking whether expression is well-formed

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

So before giving the direct definition,

we define a **formal procedure** to check whether an expression is a **well-formed expression** or not.

The procedure is as follows -

Given an expression  $e$ ,

- first check whether  $e$  is
  - $a \equiv \text{value}$ , or
  - $a \equiv \text{variable}$
 in which cases  $e$  passes the check and is a **well-formed expression**.

Failing that,

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

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

### ≡ well-formed expression

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

Recall, that last time in §1.5., when we were formally checking that  $x^3 \cdot x^5 + x^2 + 1$  is indeed a

**≡ well-formed expression**, we skipped the part about checking whether

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

which is present in the very last part of the formal procedure

**≡ checking whether mathematical expression is well-formed**.

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

the input to a function  $\in$  the domain of the function

, Thus we could potentially face mess-ups like

$$(1, 2) + 3$$

Here, the expression is not well-formed because  $(1, 2)$  is not a valid input for  $+$

(in other words  $(1, 2) \notin$  the domain of  $+$ )

, but we had no way to prevent this before.

Now, with types, this problem is solved!

If a function has type  $T1 \rightarrow T2$ ,

and Haskell wants to check whether whatever input has been given to it is a valid input or not,

it need only check that this input is of type  $T1$ .

We can see this in action with `double` -

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

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

However -

```
>>> double 'a'

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

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

This is actually the point of types, and the consequences are very powerful.

Why? Recall that  $\equiv$  **well-formed expressions** are supposed to be only those expressions which are meaningful. Since Haskell has the power to check whether expressions are well-formed or not, it will never allow us to write a “meaningless” expression.

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

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

### §3.6. Infix Binary Operators

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

$\equiv$  **using infix operator as function**

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

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

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

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

**A** using function as infix operator

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

### §3.6.1. Precedence

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

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

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

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

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

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

This problem is called the problem of “precedence”, i.e.,

“which operations in an expression are meant to be applied first (preceding) and which to be applied later?”

Haskell has a convention for handling all possible ⊕ **infix binary operators** that extends the PEMDAS convention.

(It assigns to each ⊕ **infix binary operator** a number indicating the precedence, and those with greater value of precedence are evaluated first)

But there still remains an issue -

What about `a - b - c`?

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

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

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

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

⊕ **left-associative**

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

$$x_1 \text{ ? } x_2 \text{ ? } x_3 \text{ ? } \dots \text{ ? } x_n$$

is equivalent to

$$( x_1 \text{ ? } x_2 ) \text{ ? } x_3 \text{ ? } \dots \text{ ? } x_n$$

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

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

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



```

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

```

### x order of operations

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

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

### ÷ right-associative

If an  $\div$  infix binary operator `?` is right-associative, it means that the expression

$$x_1 \text{ ? } x_2 \text{ ? } x_3 \text{ ? } \dots \text{ ? } x_{n-2} \text{ ? } x_{n-1} \text{ ? } x_n$$

is equivalent to

$$x_1 \text{ ? } x_2 \text{ ? } x_3 \text{ ? } \dots \text{ ? } x_{n-2} \text{ ? } ( x_{n-1} \text{ ? } x_n )$$

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

## §3.7. Logic

### §3.7.1. Truth

The way to represent truth or falsity in Haskell is to use the value `True` or the value `False` respectively.

Both values are of type `Bool`.

```

>>> :type True
True :: Bool

>>> :type False
False :: Bool

```

The `Bool` type means “true or false”.

The values `True` and `False` are called `Booleans`.

### §3.7.2. Statements

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

λ simplest logical statements

```
>>> 1 < 2
True

>>> 2 < 1
False

>>> 5 = 5
True

>>> 5 /= 5
False

>>> 4 = 5
False

>>> 4 /= 5
True
```

Note that `/=` is written as `/=`

Note that `<=` is written as `<=`

etc.

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

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

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

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

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

λ type of <

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

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

## §3.8. Conditions

So we can use these functions to define some “condition” on a `variable`.

For example -

λ condition on a variable

```
isLessThan5 :: Integer -> Bool
isLessThan5 x = x < 5
```

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

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

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

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

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

$$(x > 4) \ \&\& \ (x \leq 10)$$

in Haskell, since `&&` represents “AND” in Haskell.

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

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

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

So the type of `&&` is -

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

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

### §3.8.1. Logical Operators

÷ logical operator

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

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

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

For example -

- if we want to express the condition

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

, we would re-express it as

$$“x \leq 6 \text{ OR } x > 15”$$

, which could finally be expressed in Haskell as

$$(x \leq 6) \ || \ (x > 15)$$

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

- if we want to express the condition

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

, we could re-express it as

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

, which could be further re-written as

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

, which then can be expressed in Haskell as

```
not ( x < 4 )
```

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

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

### §3.8.1.1. Exclusive OR aka XOR

Finally, we define a logical operator called XOR.

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

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

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

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

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

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

## §3.9. Function Definitions

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

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

### §3.9.1. Using Expressions

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

An example -

If we want write the following definition

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

Then we can write in Haskell -

λ basic function definition

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

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

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

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

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

We can include this information in the definition -

λ function definition with explicit type

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

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

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

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

λ xor

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

## §3.9.2. Some Conveniences

### §3.9.2.1. Piecewise Functions

If we have a function definition like

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

, it can be written in Haskell as

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

For example,

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

can written in Haskell as

```
λ basic usage of guards
signum :: Double -> Double
signum x
  | x > 0 = 1
  | x == 0 = 0
  | x < 0 = -1
```

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

$$\langle \text{functionName} \rangle (x) := \left\{ \begin{array}{l} \langle \text{expression}_1 \rangle \quad ; \text{if } \langle \text{condition}_1 \rangle \\ \langle \text{expression}_2 \rangle \quad ; \text{if } \langle \text{condition}_2 \rangle \\ \langle \text{expression}_3 \rangle \quad ; \text{if } \langle \text{condition}_3 \rangle \\ \vdots \\ \langle \text{expression}_N \rangle \quad ; \text{if } \langle \text{condition}_N \rangle \\ \langle \text{expression}_{N+1} \rangle \quad ; \text{otherwise} \end{array} \right.$$

, it can be written in Haskell as

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

This `|` syntax symbol is called a “guard”.

For example -

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

If a piecewise definition has only two parts

$$\langle \text{functionName} \rangle (x) := \left\{ \begin{array}{l} \langle \text{expression}_1 \rangle \quad ; \text{if } \langle \text{condition} \rangle \\ \langle \text{expression}_2 \rangle \quad ; \text{otherwise} \end{array} \right.$$

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

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

For example -

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

### §3.9.2.2. Pattern Matching

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

#### ⚠ exhaustive pattern matching

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

We could be smarter and save some keystrokes.

#### ⚠ pattern matching

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

Another small pattern match equivalent to `xor1` -

#### ⚠ unused variables in pattern match

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

But since the variables `b1` and `b2` are not used in the right-hand side, we can replace them with `_` (read as “wildcard”)

#### ⚠ wildcard

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

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

We can use other functions to help us as well -

#### ⚠ using other functions in RHS

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

We can also piecewise definitions in a pattern match -

#### ⚠ pattern matches mixed with guards

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

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



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

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

λ trivial case

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

λ non-trivial case

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

### §3.9.2.3. Where, Let

λ where

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

λ let

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

### §3.9.2.4. Without Inputs

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

λ if-then-else example

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

We can see that this is just equivalent to

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

which can be shortened even further

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

, rewritten by  $\lambda$  using infix operator as function

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

and thus can finally be shortened to the extreme

$\lambda$  function definition without input variables

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

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

### §3.9.2.5. Anonymous Functions

An anonymous function like

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

can written as

$\lambda$  basic anonymous function

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

Note that we used  $\rightarrow$  in place of  $\mapsto$ ,

and also added a  $\backslash$  (pronounced “lambda”) before the input variable.

For an example with multiple inputs, consider

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

which can be written as

$\lambda$  multi-input anonymous function

```
( \ x y → 1/x + 1/y )
```

### $\lambda$ only nand

It is a well know fact that one can define all logical operators using only `nand`. Well, let's do so.

Redefine `and`, `or`, `not` and `xor` using only `nand`.

### §3.9.3. Recursion

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

We can use the recurrence

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

to define the factorial function.

#### λ factorial

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

We can use the standard Pascal's recurrence

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

to define the binomial or “choose” function.

#### λ binomial

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

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

#### λ naive fibonacci definition

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

## §3.10. Optimization

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

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

#### λ computation of naive fibonacci

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

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

That is what the following definition achieves.

#### λ fibonacci by tail recursion

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

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

#### ⚠ computation of tail recursion fibonacci

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

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

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

## §3.11. Numerical Functions

### ⊕ Integer and Int

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

`Integer` can hold any number no matter how big, up to the limit of your machine's memory, while `Int` corresponds to the set of positive and negative integers that can be expressed in 32 or 64 bits (based on system) with the bounds changing depending on implementation (guaranteed at least  $-2^{29}$  to  $2^{29}$ ). Going outside this range may give weird results.

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

We will assume `Integer` wherever possible.

### ⊕ Rational

`Rational` and `Double` are the types used to deal with non-integral numbers. The former is used for fractions or rationals while the latter for reals with varying amount of precision.

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

### ÷ Double

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

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

>>> sqrt 99999 :: Double
316.226184874055

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

### ÷ Integer and Int

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

`Integer` can hold any number no matter how big, up to the limit of your machine's memory, while `Int` corresponds to the set of positive and negative integers that can be expressed in 32 or 64 bits (based on system) with the bounds changing depending on implementation (guaranteed at least  $-2^{29}$  to  $2^{29}$ ). Going outside this range may give weird results.

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

We will assume `Integer` wherever possible.

### ÷ Rational

`Rational` and `Double` are the types used to deal with non-integral numbers. The former is used for fractions or rationals while the latter for reals with varying amount of precision.

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

### ÷ Double

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

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

>>> sqrt 99999 :: Double
316.226184874055

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

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

```
>>> 7 + 3
10
>>> 3 + 8
11

>>> 97 + 32
129

>>> 3 - 7
-4

>>> 5 - (-6)
11

>>> 546 - 312
234

>>> 7 * 3
21

>>> 8*4
32

>>> 45 * 97
4365

>>> 45 * (-12)
-540

>>> (-12) * (-11)
132

>>> abs 10
10

>>> abs (-10)
10
```

The internal definition of addition and subtraction is discussed in the appendix while we talk about some multiplication algorithms in chapter 10. For now, assume that these functions work exactly as expected.

**Abs** is also implemented in a very simple fashion.

λ Implementation of abs function

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

### §3.11.1. Division, A Trilogy

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

**recip** is a function which reciprocates a given number, but it has rather interesting type signature. It is only defined on types with the **Fractional** “type-class”. This refers to a lot of things, but the

most common ones are `Rational`, `Float` and `Double`. `recip`, as the name suggests, returns the reciprocal of the number taken as input. The type signature is `recip :: Fractional a => a -> a`

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

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

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

Again, notice the type signature of `(/)` is `Fractional a => a -> a -> a`.<sup>4</sup>

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

Say we want only the quotient, then we have `div` and `quot` functions.

These functions are often coupled with `mod` and `rem` are the respective remainder functions. We can get the quotient and remainder at the same time using `divMod` and `quotRem` functions. A simple example of usage is

```
>>> 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.

---

<sup>4</sup>It is worth pointing out that one could define `recip` using `(/)` as well given 1 is defined. While this is not standard, if `(/)` is defined for a data type, Haskell does automatically infer the reciprocation. So technically, for a datatype to be a member of the type class `Fractional` it needs to have either reciprocation or division defined, the other is inferred.

### x Div vs Quot

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

```
>>> 8 `div` 3
2

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

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

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

>>> 8 `quot` 3
2

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

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

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

### x Mod vs Rem

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

```
>>> 8 `mod` 3
2

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

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

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

>>> 8 `rem` 3
2

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

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

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



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

The thing here is we always want our division algorithm to satisfy  $d * q + r = n$ ,  $|r| < |d|$  where  $d$  is the divisor,  $n$  the dividend,  $q$  the quotient and  $r$  the remainder.

The issue is for any  $-d < r < 0 \Rightarrow 0 < r < d$ . This means we need to choose the sign for the remainder.

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

Basically, `div` returns the floor of the true division value (recall  $\lfloor -3.56 \rfloor = -4$ ) while `quot` returns the truncated value of the true division (recall  $\text{truncate}(-3.56) = -3$  as we are just truncating the decimal point off). The reason we keep both of them in Haskell is to be comfortable for people who come from either of these languages.

Also, The `div` function is often the more natural one to use, whereas the `quot` function corresponds to the machine instruction on modern machines, so it's somewhat more efficient (although not much, I had to go upto  $10^{100000}$  to even get millisecond difference in the two).

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

ⓧ 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 = divide n d

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

ⓧ Another Division

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

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

### §3.11.2. Exponentiation

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

**x** Can you see the difference?

What can we say about the three exponentiation operators?

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

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

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

- `^^` is raising fractional numbers to general integral powers. That is, it needs to be sure that the reciprocal of the base exists (negative powers) and doesn't throw an error if the power is negative.
- `**` is raising numbers with floating point to powers with floating point. This makes it the most general exponentiation.

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

```
>>> 2^1000
10715086071862673209484250490600018105614048117055336074 ...

>>> 2 ^^ 1000
1.0715086071862673e301

>>> 2^10000
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

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

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

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

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

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

#### ⚡ A better exponentiation algorithm using divide and conquer

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

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

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

The idea is that we are now making at most 3 multiplications at each step and there are at most  $\lceil \log_2(b) \rceil$  steps. This brings us down from  $b$  multiplications to  $3 \log(b)$  multiplications. Furthermore, most of these multiplications are somewhat balanced and hence optimized.

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

### §3.11.3. gcd and lcm

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

```
>>> :t gcd
gcd :: Integral a => a -> a -> a

>>> :t lcm
lcm :: Integral a => a -> a -> a

>>> gcd 12 30
6

>>> lcm 12 30
60
```

We will now try to define these functions ourselves.

Let's say we want to find  $g := \text{gcd}(p, q)$  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  $g$ ,  $\text{gcd}(p, q) = \text{gcd}(q, r)$ . 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

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

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

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

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

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

### §3.11.4. Dealing with Characters

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

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

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

In Haskell, the function `ord :: Char → Int` takes a character and returns its corresponding numeric code. The function `chr :: Int → Char` does the inverse: it takes a number and returns the character it represents.

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

### §3.12. Mathematical Functions

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

```
>>> sqrt 81
9.0

>>> log (2.71818)
0.9999625387017254

>>> log 4
1.3862943611198906

>>> log 100
4.605170185988092

>>> logBase 10 100
2.0

>>> exp 1
2.718281828459045

>>> exp 10
22026.465794806718

>>> pi
3.141592653589793

>>> sin pi
1.2246467991473532e-16

>>> cos pi
-1.0

>>> tan pi
-1.2246467991473532e-16

>>> asin 1
1.5707963267948966

>>> asin 1/2
0.7853981633974483

>>> acos 1
0.0

>>> atan 1
0.7853981633974483
```

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

```
>>> a = pi :: Float
>>> a
3.1415927

>>> b = pi :: Double
>>> b
3.141592653589793
```

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

### §3.12.1. Binary Search

#### ⊕ Hi-Lo game

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

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

This idea also works for slightly less direct questions:

#### ✕ Hamburgers (Codeforces 371C)

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

At the current moment, in his pantry he has:

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

And the market prices per unit is:

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

Polycarpus's wallet has  $r$  rubles.

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

Write function

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

Examples

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

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

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

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

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

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

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

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

Here is a similar exercise for your practice.



### x House of Cards (Codeforces 471C)

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

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

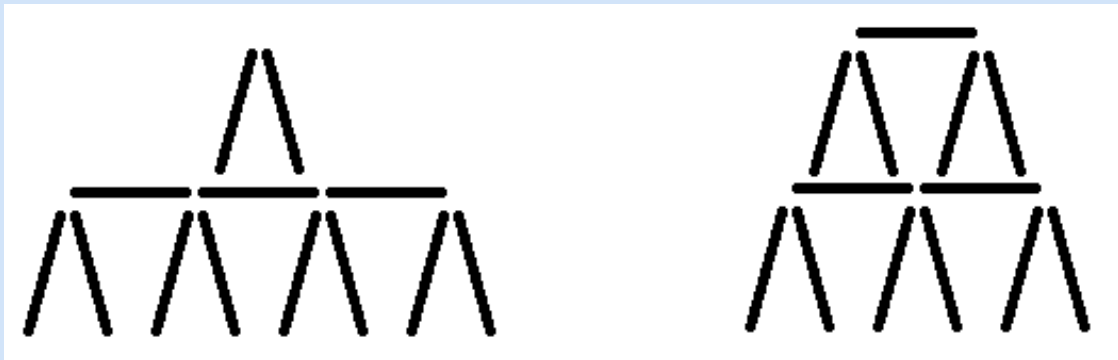
The height of the house is the number of floors in it.

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

Examples

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

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



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

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

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

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

### ⚙ Square root by binary search

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

### ✕ Cube Root

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

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

```
sqrt = bsSqrt 0.00001
```

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

## §3.12.2. Taylor Series

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

#### λ Sin and Cos using Taylor Approximation

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

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

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

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

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

#### x Inverse Trig

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

## §3.13. Exercises

#### x Collatz

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

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

## 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  $1 \times 1$  squares and one  $2 \times 2$  square.

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

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

**X Knitting Batik (COMPFEST 13, Codeforces 1575K)**

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

Define a sub-rectangle as an pair of two cells  $((x_1, y_1), (x_2, y_2))$ , denoting the top-left cell and bottom-right cell (inclusively) of a sub-rectangle in the grid. Two sub-rectangles  $((x_1, y_1), (x_2, y_2))$  and  $((x_3, y_3), (x_4, y_4))$  have the same pattern if and only if the following holds:

- (i) they have the same width ( $x_2 - x_1 = x_4 - x_3$ );
- (ii) they have the same height ( $y_2 - y_1 = y_4 - y_3$ );
- (iii) for every pair  $i, j$  such that  $0 \leq i \leq x_2 - x_1$  and  $0 \leq j \leq y_2 - y_1$ , the color of cells  $(x_1 + i, y_1 + j)$  and  $(x_3 + i, y_3 + j)$  is the same.

Write a function `countBaltik` of type

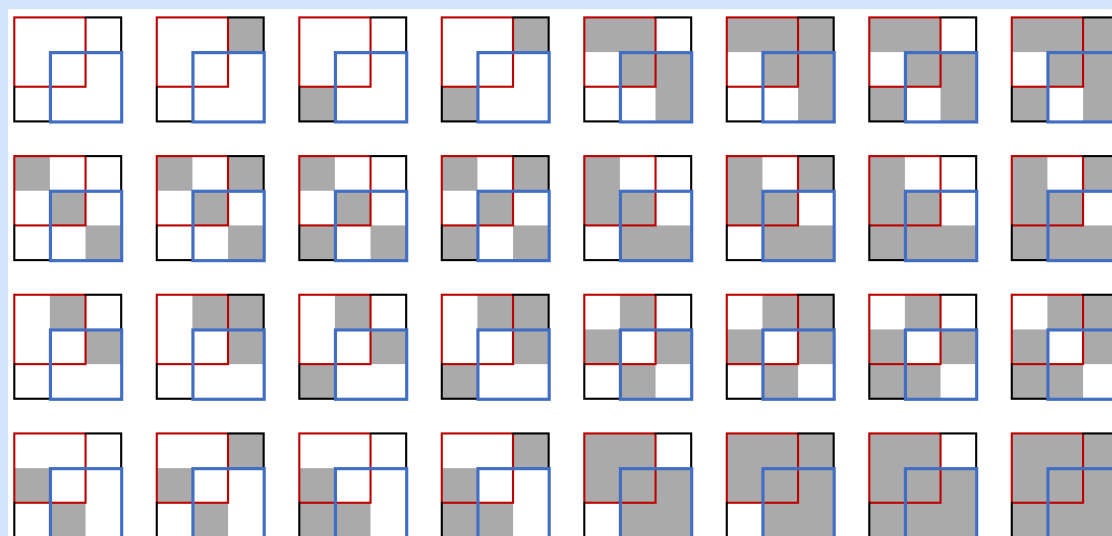
`(Int, Int) → Int → (Int, Int) → (Int, Int) → (Int, Int) → Integer` to count the number of possible batik color combinations, such that the subrectangles  $((a_x, a_y), (a_x + r - 1, a_y + c - 1))$  and  $((b_x, b_y), (b_x + r - 1, b_y + c - 1))$  have the same pattern.

Input `countBaltik` takes as input:

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

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

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



### x Modulo Inverse

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

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

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

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

### x New Bakery(Codeforces)

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

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

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

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

Example

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

Note

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

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

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



**x** Sumac

A Sumac sequence starts with two non-zero integers  $t_1$  and  $t_2$ .  
The next term,  $t_3 = t_1 - t_2$   
More generally,  $t_n = t_{n-2} - t_{n-1}$   
The sequence ends when  $t_n \leq 0$ . All values in the sequence must be positive.  
Write a function `sumac :: Int → Int → Int` to compute the length of a Sumac sequence given the initial two terms,  $t_1$  and  $t_2$ .  
Examples(Sequence is included for clarification)

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

**x** Binet Formula

Binet's formula is an explicit, closed form formula used to find the  $n$ th term of the Fibonacci sequence. It is so named because it was derived by mathematician Jacques Philippe Marie Binet, though it was already known by Abraham de Moivre.  
The problem with this remarkable formula is that it is cluttered with irrational numbers, specifically  $\sqrt{5}$ .

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

While computing using the Binet formula would only take  $2 * \log(n) + 2$  operations (exponentiation takes  $\log(n)$  time), doing so directly is out of the question as we can't represent  $\sqrt{5}$  exactly and the small errors in the approximation will accumulate due to the number of operations.  
So an idea is to do all computations on a tuple  $(a, b)$  which represents  $a + b\sqrt{5}$ . We will need to define addition, subtraction, multiplication and division on these tuples as well as define a fast exponentiation here.  
With that in hand, Write a function `fibMod :: Integer → Integer` which computes Fibonacci numbers using this method.

**x** A puzzle (UVA 10025)

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

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

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

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

with  $n = 7$

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

Examples

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

### X Rating Recalculation (Code Forces)

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

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

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

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

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

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

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

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

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

Examples

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

### X Knuth's Arrow

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

$$3 \cdot 4 = 12$$

$$3 \uparrow 4 = 3 \cdot (3 \cdot (3 \cdot 3)) = 3^4 = 81$$

$$3 \uparrow\uparrow 4 = 3 \uparrow (3 \uparrow (3 \uparrow 3)) = 3^{3^{3^3}} = 3^{7625597484987}$$

$$\approx 1.25801429062749131786039069820328121551804671431659601518967 \times 10^{3638334640024}$$

$$3 \uparrow\uparrow\uparrow 4 = 3 \uparrow\uparrow (3 \uparrow\uparrow (3 \uparrow\uparrow 3))$$

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

**X Caves (IOI 2013, P4)**

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

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

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

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

**X Carnivel (CEIO 2014)**

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

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

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