Haskell for CMI

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

To someone

Preface

§0.1. Why you should care about Haskell

§0.1.1. If You Like Programming

§0.1.1.1. Influence on Programming Language Design

Haskell pioneered features that are now mainstream:

- Type inference - now in TypeScript, Kotlin, Scala
- Algebraic data types - seen in Rust, Swift
- Monads - show up in asynchronous/await and error handling patterns
- Pattern matching - used in modern JS, Python (3.10+), and Scala

Haskell is like the **testbed** where many modern language ideas are born.

The academic research community that develops and creates the features of modern programming languages highly prefer Haskell as their conceptual base and prototyping testbed.

§0.1.1.2. Understandabilty, Readability and Refactorability

There is a common stereotype that once a large program is written and debugged, it usually gets so complicated that it is better to just change even a single letter of the code-base.

This leads to major problems when the functionality of the program needs to be expanded or changed. In the software engineering / corporate setting, this is known as the "refactorabilty" problem.

Haskell, by design, disallows and discourages this. Haskell code has many properties that make it very understandable and readable. If you have a baseline understanding of Haskell, it will not be easy to end up in a situation where you don't know what your own code is trying to do.

It also makes Haskell very readable to such an extent that it is often the case that just reading one line of a function definition is enough to understand what that function is supposed to do.

And obviously, this makes Haskell celebratedly one of the most refactorable languages.

§0.1.1.3. Makes you a Better Programmer

The concepts in Haskell which make the above things true can be applied to any language, it's just that while Haskell by its very structure forces / guides you into it, in other languages it's up to your own discipline to do so.

That means that once you learn Haskell, you will be able to apply all these concepts with their extremely useful consequences in any programming language, and thus become a better programmer.

§0.1.1.4. Catches Bugs before Running

Haskell has a thing called "types" which is purely logical, but eliminates nearly all possible bugs even before the first time that you run the program.

§0.1.1.5. Builds Safety-Critical Software

In military systems, spaceflight, and financial systems, making sure that code works as intended and that there are no bugs is a strict and extremely important necessity.

In this use-case, Haskell's properties of having nearly no bugs, being correct by construction etc. are highly prized, as evidenced by its use by -

- NASA (the American spaceflight organization)
- Galois (security software for NSA and NASA)
- Anduril (the famous military infrastructure startup)
- Cardano (a currency based on blockchain infrastructure, fully written in Haskell)
- IOHK (one of the biggest blockchain providers)
- Bitnomial (cryptocurrency options and futures firm)
- Standard Chartered (global banking firm)

§0.1.1.6. Used by Prestigious People and Organizations

Facebook uses Haskell for one of the best spam detection systems in the world.

Y Combinator is one of the largest startup advisory and venture capital funding organizations, which helped launch the likes of AirBnB, Twitch, Reddit, DropBox, GitLab and Zepto.
Its founder, Paul Graham, is a huge proponent of the functional programming paradigm (which Haskell follows) , and credits it as one of the reasons for the success of his own startup.

Jane Street is one of the most famous quantitative trading firms in the world. It uses OCaml (another functional-paradigm programming language) as the primary language and backbone of its code-base.

Haskell is very useful in analyzing, comparing and transforming textual data. That is why Pandoc, the premier software for c0nverting between file formats (word to pdf, latex to typst etc.) is written in Haskell. Semantic, the software that GitHub uses to compare different programming languages and process data for the Copilot AI is also uses Haskell.

And the list goes on…

§0.1.2. If You Like Mathematics

§0.1.2.1. Haskell is Math

Haskell is pretty much as close to math as you can get in terms of programming languages. The language is wholly based on the extremely mathematical ideas of induction, recursion, domain, co-domain, ~free-generation~, ~currying~, and ~category theory~.

Even if you don't know all of these concepts yet, what you can assume is that if your basic mathematical concepts are clear, then writing in Haskell should, in principle, not be extremely different from writing precise mathematics.

And hopefully that gets you a bit interested.

§0.1.2.2. Math begets Haskell

Haskell is wholly founded on mathematical concepts. Nearly all of the ideas it uses first appeared in the minds of various mathematicians and logicians.

So Haskell is built upon a long history and rich tradition and mathematical thought.

So you can choose to treat Haskell as a way to see how mathematical concepts can work beautifully in action and explore the real-life uses abstract math can have.

§0.1.2.3. Haskell begets Math

Haskell can actually help you learn new math and solidify you r current knowledge of math.

In this course, you will learn about -
- Precision, i.e., a deeper understanding of structure of mathematical writing
- Currying, which is extensively used throughout all of math, and you should see it in Linear Algebra.
- Types, which are a very useful mental tool for understanding and checking proofs, and can be applied to digest proofs faster and better throughout your life
- Category Theory (a little bit), which is the theory of maps from various mathematical objects to other ones

§0.1.2.4. Proof Assistants

Proof Assistants are tools that let mathematicians verify a proof beyond any doubt. They are like programming languages that you can write a formal proof in, and the computer will verify whether the proof works or not.

This has become an interesting topic in the discussion about the future of mathematics, and many famous mathematicians (such as Terrence Tao) believe that proof assistants will be an indispensable tool for the progress of mathematics.

As many proof assistants, such as Agda, Coq, and Lean can be thought of as extension based on or built up from the concepts used in Haskell, knowing Haskell should provide advantage in understanding how to write proofs using such assistants.

§0.1.2.5. Used in Mathematical Computation

Haskell is used in computing various mathematical objects and to carry out mathematical experiments which test some statement by generating examples or counterexamples.

This generated data can be used to disprove mathematical conjectures or make new ones.

§0.1.2.6. Expressivity of Haskell

You might surprised by the depth and breadth of mathematical objects that can be defined in Haskell.

For example, one can define a list of ALL the prime numbers.

§0.2. How to Learn Haskell

§0.2.1. If You Like Math

If you follow this book, you will see that Haskell follows a mathematical style as closely as it can. So, if you can learn to express a few mathematical ideas precisely enough, Haskell *should* become child's play.

In order to test your knowledge and be sure that you are learning the correct meanings of things, it is encouraged that you try coding all the assignments, graded or otherwise. Also, it should be helpful to practice programming by getting problems from other sources as well.

The beauty of practicing programming is that one can check their solution by *running* a few examples, so you can get lots of practice and *verify it by yourself* too.

But most importantly, remember to have fun!

§0.2.2. If You Like Programming

If you have dabbled in the functional paradigm before, you know what you are getting into.

But if the languages you have dealt with in the past have been mainly procedural, object-oriented, etc., then you need to read these next sections VERY CAREFULLY.

Haskell is NOT a programming language in any sense of "programming language" that you have encountered before. Often when some concept comes up that looks or feels similar to programming you might have done in the past, please be very careful to not jump to assumptions. The functional paradigm rarely aligns with any of the concepts of other paradigms.

Possibly the best way to avoid such assumptions would be to -

**Just think of Haskell as a way of writing mathematical expressions,
which only happens to be runnable by mere coincidence.**

Basically, treat Haskell code as more of math notation than programming syntax, and you should be fine.

Keep your mind open, and you might just learn something that changes your entire perspective on programming for the better.

But most importantly, have fun!

§0.3. How to Read this Book

Definitions look like this.

x **Exercise**

Exercises look like this.

```
Code looks like this.
```

You can use the link "Contents" in the upper-left corner of every page to jump back to the Table of Contents.

Many things, especially coloured text, are links. You can exploit that as you wish.

Happy reading!

Table of Contents

Preface iii

§0.1. Why you should care about Haskell iii

§0.1.1. If You Like Programming iii

§0.1.1.1. Influence on Programming Language Design iii

§0.1.1.2. Understandabilty, Readability and Refactorability iii

§0.1.1.3. Makes you a Better Programmer iii

§0.1.1.4. Catches Bugs before Running iii

§0.1.1.5. Builds Safety-Critical Software iv

§0.1.1.6. Used by Prestigious People and Organizations iv

§0.1.2. If You Like Mathematics iv

§0.1.2.1. Haskell is Math iv

§0.1.2.2. Math begets Haskell iv

§0.1.2.3. Haskell begets Math v

§0.1.2.4. Proof Assistants v

§0.1.2.5. Used in Mathematical Computation v

§0.1.2.6. Expressivity of Haskell v

§0.2. How to Learn Haskell v

§0.2.1. If You Like Math v

§0.2.2. If You Like Programming v

§0.3. How to Read this Book vi

Table of Contents vii

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 6
§ 1.6.2.3. Piecewise Functions 6
§ 1.6.2.4. Pattern Matching 7
§ 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 **24**

§ 2.1. Installation 24

§ 2.1.1. General Instructions 24

§ 2.1.2. Choose your Operating System 24

§ 2.1.2.1. Linux 24

§ 2.1.2.2. MacOS 25

§ 2.1.2.3. Windows 26

§ 2.2. Running Haskell 26

§ 2.3. Fixing Errors 27

§ 2.4. Autocomplete 27

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
  - $x^3 \cdot x^5 + x^2 + 1$
  - $f(g(x, y), f(a, h(v), c), h(h(h(n))))$
  - $\forall \alpha \in \mathbb{R}\ \exists L \in \mathbb{R}\ \forall \varepsilon > 0\ \exists \delta > 0\ \ |\ x - \alpha\ | < \delta \Rightarrow |\ f(x) - f(\alpha)\ | < \varepsilon$

  etc.

§ 1.3. Values

mathematical value

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

The following examples should clarify further.

Examples include -

- The real number π
- The order $<$ on $\mathbb{N}$

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

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

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

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

§ 1.4. Variables

÷ **mathematical variable**

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

The following examples should clarify further.

For example, consider the following function definition -

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

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

Consider the following theorem -

Theorem Adding 1 to a natural number makes it bigger.

Proof Take n to be an arbitrary natural number.

We know that $1 > 0$.
Adding n to both sides of the preceding inequality yields

$$n + 1 > n$$

Hence Proved !! ■

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

Here is another theorem -

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

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

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

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

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

Hence Proved! ■

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

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

§ 1.5. Well-Formed Expressions

Consider the expression -

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

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

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

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

checking whether mathematical expression is well-formed

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

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

The procedure is as follows -

Given an expression e,

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

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

Failing that,

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

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

well-formed mathematical expression

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

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

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

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

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

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

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

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

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

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

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

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

Done!

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

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

Check whether the expression

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

is well-formed or not.

§ 1.6. Function Definitions

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

§ 1.6.1. Using Expressions

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

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

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 ÷ **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 -

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

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

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

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".

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

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

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

§ 1.6.2.2. Anonymous Functions

A function definition such as

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

for convenience, can be rewritten as -

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

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

Also, we used $\mapsto$ in place of $:=$

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

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

Consider -

$$\text{harmonicSum} : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \to \mathbb{R}_{>0}$$
$$\text{harmonicSum}(x, y) := \frac{1}{x} + \frac{1}{y}$$

which, for convenience, can be rewritten as -

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

§ 1.6.2.3. Piecewise Functions

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

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

For example, consider the following definition -

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

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

§ 1.6.2.4. Pattern Matching

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

For example, consider the last definition -

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

which can be rewritten as -

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

This definition relies on checking the form of the input.

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

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

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

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

§ 1.6.3. Recursion

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

For example, consider defining the famous fibonacci function -

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

§ 1.6.3.1. Termination

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

For example, consider the following definition -

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

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

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

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

termination of recursive definition

In general, a recursive definition is said to **terminate on an input**
if and only if
it eventually gives an ***actual specific output for that input***.

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

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

§ 1.6.3.2. Induction

principle of mathematical induction

Suppose we have an infinite sequence of statements $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots$
and we can prove the following 2 statements -
- φ_0 is true
- For each $n > 0$, if φ_{n-1} is true, then φ_n is also true.

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

The above definition should be read as follows, given a sequence of formulas:
- The first one is true.
- Any formula being true, implies that the next one in the sequence is true.

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

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 $P_1, P_2, \cdots, P_{n+1}$. By the induction hypothesis, the n people $P_1, P_2, \cdots, P_n$ all have the same height. Similarly the n people $P_2, P_3, \cdots, P_{n+1}$ all have the same height. Both groups of people contain $P_2, P_3, \cdots, P_n$, so P_1 and P_{n+1} have the same height as $P_2, P_3, \cdots, P_n$. Thus all of $P_1, P_2, \cdots, P_{n+1}$ have the same height. Hence by induction, for any n any group of n people have the same height. Letting n be the total number of people in the world, we conclude that all people have the same height. Is there a flaw in this argument?

proving the principle of induction

Prove that the following statements are equivalent -

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

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

§1.6.3.3. Proving Termination using Induction

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

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

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

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

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

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

  Hence φ_n is proved!

Hence the theorem is proved!! ∎

§1.7. Infix Binary Operators

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

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

$$x + y$$

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

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

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

÷ **infix binary operator**

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

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

§ 1.8. Trees

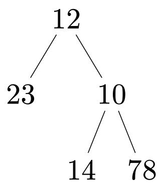
Trees are a way to structure a collection of objects.

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

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

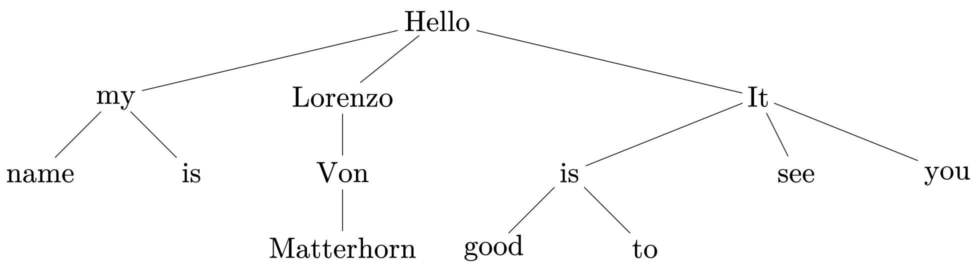
§ 1.8.1. Examples of Trees

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



The line segments are what defines the structure.

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



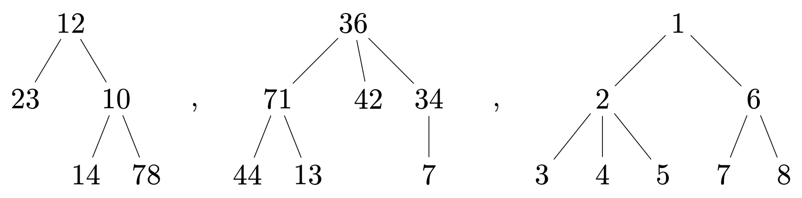
§ 1.8.2. Making Larger Trees from Smaller Trees

If we have an object -

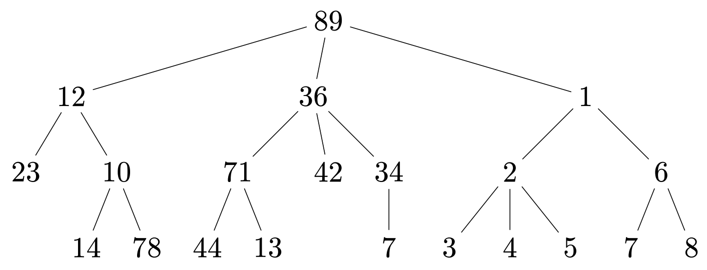
$$89$$

and a few trees -

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



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



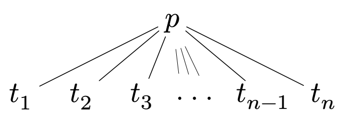
In general, if we have an object

$$p$$

and a bunch of trees

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

, 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.

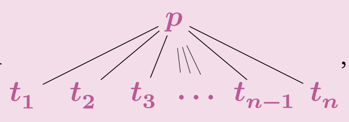
checking whether object is tree

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**

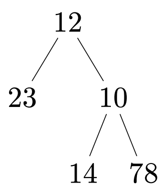
Failing that,

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

tree

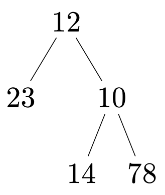
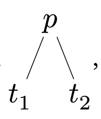
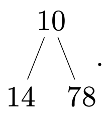
Given a set S, a ***mathematical object*** is said to be a **tree over S** if and only if it passes the formal checking procedure defined in **checking whether object is tree**.

Let us use this definition to check whether

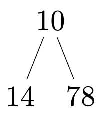


is a **tree over the natural numbers**.

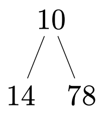
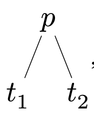
Let's start -

 is of the form , where p is 12 , t_1 is 23 , and t_2 is .

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.

 is of the form , where p is 10 , t_1 is 14, and t_2 is 78

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

§ 1.8.4. Structural Induction

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

÷ **structural induction for trees**

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

If we can prove the following two statements -

- For each $s \in S$, φ_s is true
- For each tree T of the form $$\begin{tikzcd}[column sep=small] & & & p \arrow[dlll, no head] \arrow[dll, no head] \arrow[dl, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\ t_1 & t_2 & t_3 & \cdots & t_{n-1} & t_n \end{tikzcd},$$

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

  then φ_T is also true.

then φ_t is true for all trees t over S.

§ 1.8.5. Structural Recursion

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

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

- either of the form $s \in S$
- or of the form $$\begin{tikzcd}[column sep=small] & & & p \arrow[dlll, no head] \arrow[dll, no head] \arrow[dl, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\ t_1 & t_2 & t_3 & \cdots & t_{n-1} & t_n \end{tikzcd}$$

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

÷ **tree size**

Let's use this principle to define the function

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

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

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

$$\text{size}\left(\begin{tikzcd}[column sep=small] & & & p \arrow[dlll, no head] \arrow[dll, no head] \arrow[dl, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\ t_1 & t_2 & t_3 & \cdots & t_{n-1} & t_n \end{tikzcd}\right) := 1 + \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + \ldots + \text{size}(t_{n-1}) + \text{size}(t_n)$$

§ 1.8.6. Termination

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

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

Proof For each tree t, let φ_t be the statement

“ The definition of size(t) terminates “

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

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

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

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

  Hence φ_T is proved!

Hence the theorem is proved!! ■

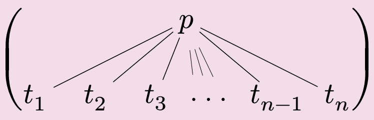
tree depth

Fix a set S.

tree depth

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

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

depth  $:= 1 + \max\limits_{1 \le i \le n} \{\text{depth}(t_i)\}$

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

Exercise

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

Using the **principle of mathematical induction**,

prove **structural induction for trees**.

§ 1.9. Why Trees?

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

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

§ 1.9.1. The Problem

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

How can we do this?

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

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

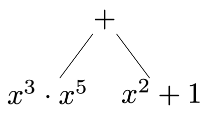
So, for dumb Haskell to do this (figure out the values of expressions, which is quite an important ability) , a vital requirement is to be able to easily put expressions in the form $f(x_1, x_2, x_3, ..., x_{n-1}, x_n)$.

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

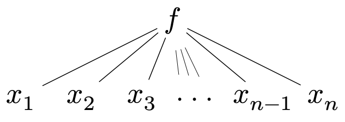
One way to make this easier to represent the expression in the form of a tree -

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

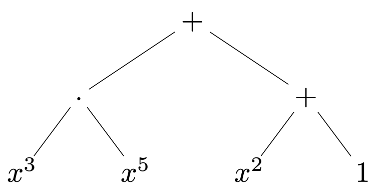


, it becomes obvious what the function is and what the inputs are to which it is applied.

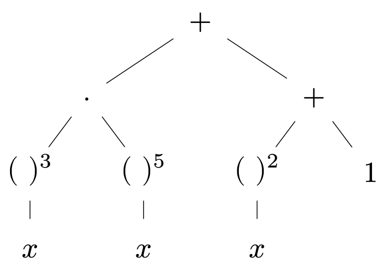
In general, we can represent the expression $f(x_1, x_2, x_3, ..., x_{n-1}, x_n)$ as



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



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



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

abstract syntax tree

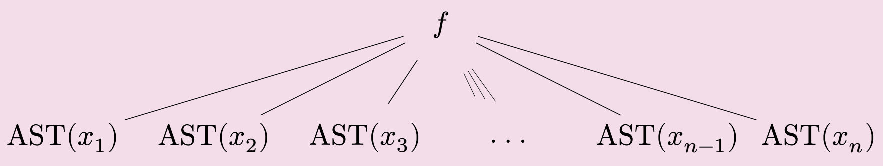
The **abstract syntax tree of a well-formed expression** is defined by applying the "function" **AST** to the expression.

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

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

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

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



§ 1.9.3. Exercises

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

Turbo The Snail(IMO 2024, P5)

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

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

[3]Atleast one author is of the opinion:

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

x Points in Triangle

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

x Joining Points(IOI 2006, 6)

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

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

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

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

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

x Usmions(USA TST 2015, simplified)

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

x Battery

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

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

x Seven Tries (Russia 2000)

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

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

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

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

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

For Example

Input1 : 6 5

Output1 : 6 7

Input2 : 8 16

Output2 : 35 53

X Rope

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

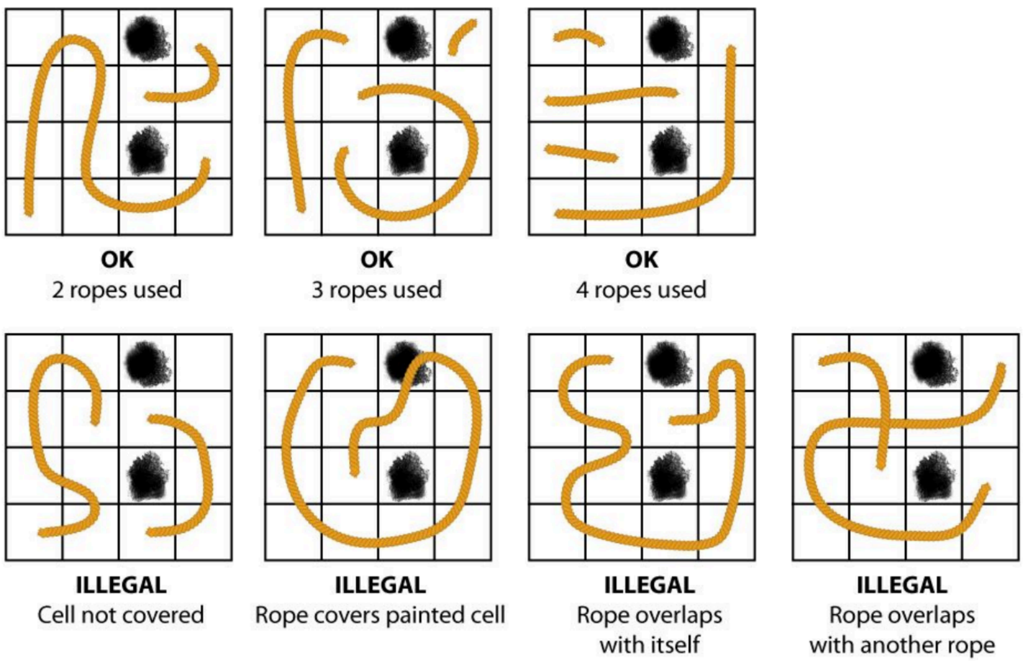


Figure out an algorithm/recipie to covering the grid using $n+1$ ropes leagally.

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

X n composite

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

X Divided by 5^n

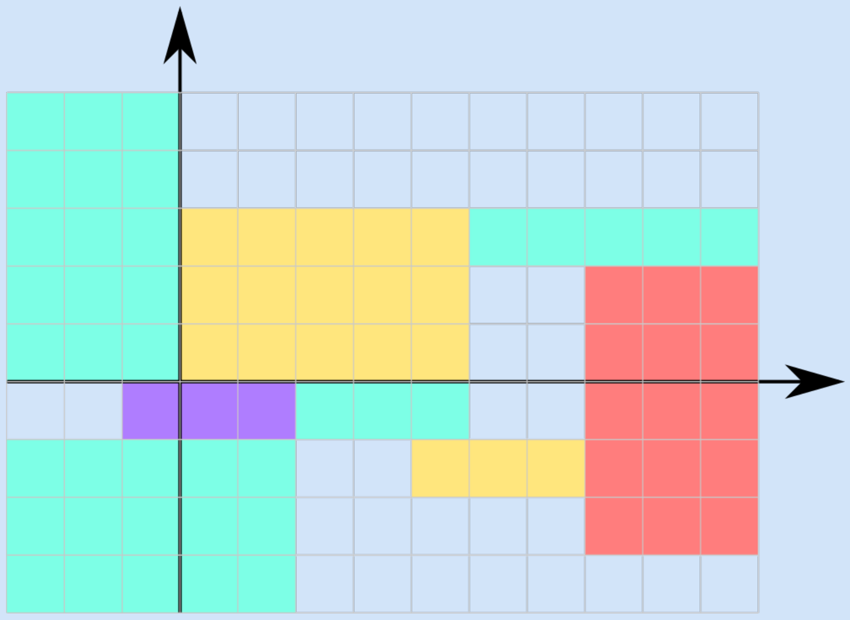
Prove that for every positive integer n, there exists an n-digit number divisible by 5^n, all of whose digits are odd.

X This was rated 2100? (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.

Seating

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

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

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

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

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

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

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

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

Yet some more Fibonnaci Identity

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

(i) Prove that

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

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

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

(iii) Evaluate

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

Round Robin

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

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 milt 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 $\frac{2+\left(-\frac{1}{2}\right)^n}{3}$

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.

X Caves (IOI 2013, P4)

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

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

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

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

X Carnivel (CEIO 2014)

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

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

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

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.