

Haskell for CMI

Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

© 2025 Ryan Hota, Shubh Sharma, Arjun Maneesh Agarwal

Text licensed under CC-by-SA-4.0

Code licensed under AGPL-3.0

This is (still!) an incomplete draft.

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

Last updated August 17, 2025.

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. Understandability, 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 “refactorability” 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 converting 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 your 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 be 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.

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.

 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. Understandability, 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
\Rightarrow mathematical value	1
§ 1.4. Variables	2
\Rightarrow 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
	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

Types as Sets	69
§ 4.1. Sets	69
\doteq set	69
\doteq empty set	69
\doteq singleton set	69
\doteq belongs	69
\doteq union	69
\doteq intersection	69
\doteq cartesian product	69
\doteq set exponent	70
§ 4.2. Types	70
§ 4.2.1. $::$ is analogous to \in or \doteq belongs	70
λ declaration of x	70
λ declaration of y	70
§ 4.2.2. $A \rightarrow B$ is analogous to B^A or \doteq set exponent	70
λ function	71
λ another function	71
§ 4.2.3. (A, B) is analogous to $A \times B$ or \doteq cartesian product	71
λ type of a pair	71
λ elements of a product type	71
λ first component of a pair	71
λ second component of a pair	72
λ function from a product type	72
λ another function from a product type	72
λ function to a product type	72
§ 4.2.4. $()$ is analogous to \doteq singleton set	72
λ elements of unit type	72
§ 4.2.5. No \doteq intersection of Types	73
§ 4.2.6. No \doteq union of Types	73
§ 4.2.7. Disjoint Union of Sets	73
\doteq disjoint union	73
§ 4.2.8. $\text{Either } A \ B$ is analogous to $A \sqcup B$ or \doteq disjoint union	73
λ elements of an either type	74

	λ function to an either type	74
	λ function from an either type	75
	λ another function from an either type	75
§ 4.2.9.	The Maybe Type	76
	λ naive reciprocal	76
	λ reciprocal using either	76
	λ function to a maybe type	77
	λ elements of a maybe type	77
	λ function from a maybe type	78
§ 4.2.10.	Void is analogous to {} or ∅ empty set	78
§ 4.3.	Currying	78
§ 4.3.1.	In Haskell	80
	∅ currying rule	80
	λ currying usage	80
	λ another currying usage	81
§ 4.3.2.	Understanding through Associativity	81
§ 4.3.2.1.	Of →	81
§ 4.3.2.2.	Of Function Application	82
§ 4.3.2.3.	Operator Currying Rule	82
	∅ operator currying rule	83
	λ operator currying usage	83
	λ another operator currying usage	84
§ 4.3.3.	Proof of the Currying Theorem	84
§ 4.4.	Exercises	86
	∅ Unital Operator	87
	∅ Shelf	88
	∅ Rack	88
	∅ Quandle	88
	∅ Kei	88
	Introduction to Lists	90
§ 5.1.	Type of List	90
§ 5.2.	Creating Lists	90
§ 5.2.1.	Empty List	90
§ 5.2.2.	Arithmetic Progression	90

	λ arithmetic progression syntax	91
	λ non-number arithmetic progressions	91
§5.3.	Functions on Lists	91
§5.4.	List Comprehension	91
§5.4.1.	Cons or <code>(:)</code>	92
	λ pattern matching lists	92
§5.5.	Length	92
	λ length of list	92
§5.5.1.	Concatenate or <code>(++)</code>	93
	λ concatenation of lists	93
§5.5.2.	Head and Tail	93
	λ head of list	93
	λ tail of list	94
	λ uncons of list	94
§5.5.3.	Take and Drop	94
	λ take from list	95
	λ drop from list	95
§5.5.4.	<code>(!!)</code>	96
§5.5.5.	List <code>→ Bool</code>	96
§5.5.5.1.	Elem	96
§5.5.5.2.	Generalized Logical Operators	97
	λ and	97
	λ or	98
§5.6.	Strings	98
§5.7.	Structural Induction for Lists	100
	≡ structural induction for lists	100
§5.8.	Sorting	101
	≡ sorted list	101
	≡ sorting	102
	λ sort	103
§5.9.	Optimization	104
	λ naive reverse	104
	λ optimized reverse	105
	λ naive splitAt	105
	λ optimized splitAt	105
§5.10.	Lists as Syntax Trees	106

§ 5.11.	Dark Magic	107
§ 5.11.1.	Exercises	108
	Polymorphism and Higher Order Functions	114
§ 6.1.	Polymorphism	114
§ 6.1.1.	Classification has always been about <i>shape</i> and <i>behaviour</i> anyway	114
	λ squaring all elements of a list	115
	λ and	115
	≡ Polymorphism	116
	λ drop	117
	≡ Behaviour	117
	≡ 2 Types of Polymorphism	117
§ 6.1.2.	A Taste of Type Classes	117
	λ Function Extensionality	118
	≡ Typeclasses	118
	λ elem	119
§ 6.2.	Higher Order Functions	119
	≡ Higher Order Functions	119
§ 6.2.1.	Currying	119
	λ curry and uncurry	121
§ 6.2.2.	Functions on Functions	121
	λ composition	121
	λ function application function	122
	λ operator precedence	122
§ 6.2.3.	A Short Note on Type Inference	122
§ 6.2.4.	Higher Order Functions on Maybe Type : A Case Study	123
	λ maybeMap	124
	Advanced List Operations	127
§ 7.1.	List Comprehensions	127
	λ Defining map using pattern matching and list comprehension	127
	λ Defining filter using pattern matching and list comprehension	127
	λ Defining cartesian product using pattern matching and list comprehension	128

	λ A naive way to get pythagorian triplets	128
	λ A mid way to get pythagorian triplets	128
	λ The optimal way to get pythagorian triplets .	128
	λ The merge function of mergesort	129
	λ An implementation of mergesort	130
	λ An implementation of Quick Sort	131
§ 7.2.	Zip it up!	132
	λ Implementation of zip function	133
	λ Implementation of zipWith function	133
	λ The zipWith fibonnaci	133
§ 7.3.	Folding, Scaning and The Gate to True Powers	136
§ 7.3.1.	Orgami of Code!	136
	λ Definition of foldr	137
	λ Definition of foldr1	137
	λ Definition of foldl and foldl1	139
	λ Implementation of unfoldr	140
	λ list of primes using unfoldr	142
	λ Space to write the definition of sublists	143
§ 7.3.2.	Numerical Integration	145
	λ Naive Integration	145
	λ Naive Integration without repeated computation	145
	λ An optimized function for numerical integration	147
§ 7.3.3.	Time to Scan	147
	≡ Scans	148
	≡ Segmented Scan	151
§ 7.4.	Excercises	152
	Introduction to Datatypes	168
§ 8.1.	Datatypes (Once Again)	168
	≡ Types 1	168
	≡ Types 2	168
§ 8.2.	Finite Types	169
	λ finite types	169
§ 8.3.	Product Types	170

§ 8.4.	Parametric Types	171
§ 8.5.	Sum Types	171
§ 8.6.	Inductive Types	172
§ 8.6.1.	Inductive Types (as a Mathematician)	172
	≡ Freely Generated Sets	173
§ 8.6.1.1.	Natural Numbers as Inductive Types	173
	λ nat	174
	λ nat and integer	174
§ 8.6.1.2.	Lists as Inductive Types	174
	λ list	175
§ 8.6.2.	(Not Quite) Inductive Types (as a Programmer)	175
§ 8.6.2.1.	Calculator	175
	λ expression	176
	λ expr example	176
§ 8.6.2.2.	Trees as Inductive Types	177
	λ tree	177
§ 8.6.2.3.	Binary Trees	178
	λ Btree	178
	λ Btree ex	178
§ 8.6.2.4.	Addressing the “Not Quite”	179
§ 8.7.	Same Same but Different	181
§ 8.7.1.	Same Same	181
	λ type aliases	181
§ 8.7.2.	Different	181

Computation as Reduction	183
--------------------------------	-----

Complexity	184
------------------	-----

§ 10.1.	Introduction	184
§ 10.2.	Asymptotics	184
§ 10.2.1.	Big El	184
	≡ Big Ell notation	184
§ 10.2.2.	The Hierarchy of Functions	185
	≡ Asymptotic Dominance	185

⊖ Representative Function wrt asymptotic dominance	186
§ 10.2.3. Big Oh notation	187
⊖ Big Oh (from Big Ell)	187
⊖ Big Oh (from hierarchy)	187
⊖ Big Oh (limit)	187
⊖ Big Oh (Classical)	188
⊖ Bachman-Landau Notation	188
§ 10.3. Asymptotic Mathematics	189
⊖ Derivative with big Oh	190
⊖ Binoial Exapansion with Big Oh	190
⊖ Taylor Series with Big Oh	190
⊖ Prime Number Theorem	190
⊖ Stirling Approximation	191
⊖ Binomial Coefficient	191
§ 10.4. Analysis of Algorithms	194
§ 10.4.1. Sorting	195
§ 10.4.1.1. Counting Sort	197
λ Counting Sort	197
§ 10.4.1.2. Radix Sort	198
λ Counting Sort With Keys	198
⊖ Radix Sort	198
§ 10.4.1.3. Survey of Sorting Algorithms	199
§ 10.5. RAM Model and Asymptotic Analysis	200
⊖ Random Access Machine	201
§ 10.5.1. What Big O doesn't want you to know?	201
⊖ Galactic Algorithms	202
§ 10.6. An Informal Survey of Multiplication Algorithms	202
λ Addition and Subtraction of two numbers	203
λ Singluar Digit Multiplication	203
λ School Book Multiplication	204
λ Naive Divide and Conquor defination	204
λ Karatsuba Multiplication algortithm	205
Advanced Data Structures	208
§ 11.1. Datatypes (One last time)	208

§ 11.2.	Stacks and Queues	208
	⊘ Containers	208
§ 11.2.1.	Stack	208
	⊘ Stack	208
§ 11.2.2.	Queue	213
	⊘ Queue	213
§ 11.2.2.1.	Amortization	214
	⊘ Ammortization	214
	⊘ Ammortized Analysis	215
	⊘ Piggy Bank Method	216
§ 11.2.2.2.	The True Queue	216
§ 11.3.	Binary Search Tree	217
	⊘ Binary Search Tree	218
§ 11.4.	Sets and Maps	222
	⊘ Sets	222
§ 11.5.	Exercise	222
	 Type Classes	223
§ 12.1.	Basics of Typeclasses	223
§ 12.1.1.	Vector Spaces	223
	λ definition of VectorSpace	223
	λ example instance of VectorSpace	224
	λ scalar field is VectorSpace	224
§ 12.1.2.	Other Typeclasses	224
	λ definition of Eq	224
	λ Colour is an Eq space	225
§ 12.1.3.	General Definition	225
	⊘ typeclass	225
	⊘ method	226
	λ defining typeclass syntax	226
	λ making a type an instance of typeclass	226
§ 12.2.	The Point of Typeclasses	227
§ 12.2.1.	The Point of Vector Spaces	227
§ 12.2.2.	The Point of any Class	227
	λ vectorSubtraction	227
	⊘ ad-hoc polymorphism	228

§ 12.2.2.1.	Types	228
§ 12.3.	Induced Instances	228
§ 12.4.	The deriving Keyword	229
§ 12.5.	Superclasses and Subclasses	229
	\equiv subclass	229
§ 12.6.	Laws	230
	\equiv typeclass laws	231
Monads		232
Appendix		233
§ 14.1.	Preface	233
§ 14.2.	Why you should care about Haskell	233
§ 14.2.1.	If You Like Programming	233
§ 14.2.1.1.	Influence on Programming Language Design	233
§ 14.2.1.2.	Understandability, Readability and Refactorability	233
§ 14.2.1.3.	Makes you a Better Programmer	233
§ 14.2.1.4.	Catches Bugs before Running	233
§ 14.2.1.5.	Builds Safety-Critical Software	233
§ 14.2.1.6.	Used by Prestigious People and Organizations	233
§ 14.2.2.	If You Like Mathematics	233
§ 14.2.2.1.	Haskell is Math	233
§ 14.2.2.2.	Math begets Haskell	233
§ 14.2.2.3.	Haskell begets Math	233
§ 14.2.2.4.	Proof Assistants	233
§ 14.2.2.5.	Used in Mathematical Computation	234
§ 14.2.2.6.	Expressivity of Haskell	234
§ 14.3.	How to Learn Haskell	234
§ 14.3.1.	If You Like Math	234
§ 14.3.2.	If You Like Programming	234
§ 14.4.	How to Read this Book	234
§ 14.5.	Table of Contents	234
§ 14.6.	Basic Theory	234
§ 14.7.	Precise Communication	234
§ 14.8.	The Building Blocks	234

§ 14.9.	Values	234
§ 14.10.	Variables	234
§ 14.11.	Well-Formed Expressions	234
§ 14.12.	Function Definitions	234
§ 14.12.1.	Using Expressions	234
§ 14.12.2.	Some Conveniences	235
§ 14.12.2.1.	Where, Let	235
§ 14.12.2.2.	Anonymous Functions	235
§ 14.12.2.3.	Piecewise Functions	235
§ 14.12.2.4.	Pattern Matching	235
§ 14.12.3.	Recursion	235
§ 14.12.3.1.	Termination	235
§ 14.12.3.2.	Induction	235
§ 14.12.3.3.	Proving Termination using Induction	235
§ 14.13.	Infix Binary Operators	235
§ 14.14.	Trees	235
§ 14.14.1.	Examples of Trees	235
§ 14.14.2.	Making Larger Trees from Smaller Trees	235
§ 14.14.3.	Formal Definition of Trees	235
§ 14.14.4.	Structural Induction	235
§ 14.14.5.	Structural Recursion	236
§ 14.14.6.	Termination	236
§ 14.15.	Why Trees?	236
§ 14.15.1.	The Problem	236
§ 14.15.2.	The Solution	236
§ 14.15.3.	Exercises	236
§ 14.16.	Installing Haskell	236
§ 14.17.	Installation	236
§ 14.17.1.	General Instructions	236
§ 14.17.2.	Choose your Operating System	236
§ 14.17.2.1.	Linux	236
§ 14.17.2.2.	MacOS	236
§ 14.17.2.3.	Windows	236
§ 14.18.	Running Haskell	236
§ 14.19.	Fixing Errors	236
§ 14.20.	Autocomplete	237

§ 14.21. Basic Syntax	237
§ 14.22. The Building Blocks	237
§ 14.23. Values	237
§ 14.24. Variables	237
§ 14.25. Types	237
§ 14.25.1. Using GHCi to get Types	237
§ 14.25.2. Types of Functions	237
§ 14.26. Well-Formed Expressions	237
§ 14.27. Infix Binary Operators	237
§ 14.27.1. Precedence	237
§ 14.28. Logic	237
§ 14.28.1. Truth	237
§ 14.28.2. Statements	237
§ 14.29. Conditions	237
§ 14.29.1. Logical Operators	238
§ 14.29.1.1. Exclusive OR aka XOR	238
§ 14.30. Function Definitions	238
§ 14.30.1. Using Expressions	238
§ 14.30.2. Some Conveniences	238
§ 14.30.2.1. Piecewise Functions	238
§ 14.30.2.2. Pattern Matching	238
§ 14.30.2.3. Where, Let	238
§ 14.30.2.4. Without Inputs	238
§ 14.30.2.5. Anonymous Functions	238
§ 14.30.3. Recursion	238
§ 14.31. Optimization	238
§ 14.32. Numerical Functions	238
§ 14.32.1. Division, A Trilogy	238
§ 14.32.2. Exponentiation	238
§ 14.32.3. <code>gcd</code> and <code>lcm</code>	238
§ 14.32.4. Dealing with Characters	239
§ 14.33. Mathematical Functions	239
§ 14.33.1. Binary Search	239
§ 14.33.2. Taylor Series	239
§ 14.34. Exercises	239
§ 14.35. Types as Sets	239

§14.36. Sets	239
§14.37. Types	239
§14.37.1. <code>::</code> is analogous to \in or \ni belongs	239
§14.37.2. <code>A → B</code> is analogous to B^A or \ni set exponent	239
§14.37.3. <code>(A , B)</code> is analogous to $A \times B$ or \ni cartesian product	239
§14.37.4. <code>()</code> is analogous to \ni singleton set	239
§14.37.5. No \ni intersection of Types	239
§14.37.6. No \ni union of Types	239
§14.37.7. Disjoint Union of Sets	240
§14.37.8. <code>Either A B</code> is analogous to $A \sqcup B$ or \ni disjoint union	240
§14.37.9. The <code>Maybe</code> Type	240
§14.37.10. <code>Void</code> is analogous to $\{\}$ or \ni empty set	240
§14.38. Currying	240
§14.38.1. In Haskell	240
§14.38.2. Understanding through Associativity	240
§14.38.2.1. Of <code>→</code>	240
§14.38.2.2. Of Function Application	240
§14.38.2.3. Operator Currying Rule	240
§14.38.3. Proof of the Currying Theorem	240
§14.39. Exercises	240
§14.40. Introduction to Lists	240
§14.41. Type of List	240
§14.42. Creating Lists	240
§14.42.1. Empty List	241
§14.42.2. Arithmetic Progression	241
§14.43. Functions on Lists	241
§14.44. List Comprehension	241
§14.44.1. Cons or <code>(:)</code>	241
§14.45. Length	241
§14.45.1. Concatenate or <code>(++)</code>	241
§14.45.2. Head and Tail	241
§14.45.3. Take and Drop	241
§14.45.4. <code>(!!)</code>	241

§ 14.45.5. List \rightarrow Bool	241
§ 14.45.5.1. Elem	241
§ 14.45.5.2. Generalized Logical Operators	241
§ 14.46. Strings	241
§ 14.47. Structural Induction for Lists	241
§ 14.48. Sorting	242
§ 14.49. Optimization	242
§ 14.50. Lists as Syntax Trees	242
§ 14.51. Dark Magic	242
§ 14.51.1. Exercises	242
§ 14.52. Polymorphism and Higher Order Functions	242
§ 14.53. Polymorphism	242
§ 14.53.1. Classification has always been about <i>shape</i> and <i>behaviour</i> anyway.	242
§ 14.53.2. A Taste of Type Classes	242
§ 14.54. Higher Order Functions	242
§ 14.54.1. Currying	242
§ 14.54.2. Functions on Functions	242
§ 14.54.3. A Short Note on Type Inference	242
§ 14.54.4. Higher Order Functions on Maybe Type : A Case Study	242
§ 14.55. Advanced List Operations	242
§ 14.56. List Comprehensions	243
§ 14.57. Zip it up!	243
§ 14.58. Folding, Scanning and The Gate to True Powers	243
§ 14.58.1. Orgami of Code!	243
§ 14.58.2. Numerical Integration	243
§ 14.58.3. Time to Scan	243
§ 14.59. Exercises	243
§ 14.60. Introduction to Datatypes	243
§ 14.61. Datatypes (Once Again)	243
§ 14.62. Finite Types	243
§ 14.63. Product Types	243
§ 14.64. Parametric Types	243
§ 14.65. Sum Types	243
§ 14.66. Inductive Types	243
§ 14.66.1. Inductive Types (as a Mathematician)	243
§ 14.66.1.1. Natural Numbers as Inductive Types	244

§ 14.66.1.2. Lists as Inductive Types	244
§ 14.66.2. (Not Quite) Inductive Types (as a Programmer)	244
§ 14.66.2.1. Calculator	244
§ 14.66.2.2. Trees as Inductive Types	244
§ 14.66.2.3. Binary Trees	244
§ 14.66.2.4. Addressing the “Not Quite”	244
§ 14.67. Same Same but Different	244
§ 14.67.1. Same Same	244
§ 14.67.2. Different	244
§ 14.68. Computation as Reduction	244
§ 14.69. Complexity	244
§ 14.70. Introduction	244
§ 14.71. Asymptotics	244
§ 14.71.1. Big El	244
§ 14.71.2. The Hierarchy of Functions	244
§ 14.71.3. Big Oh notation	245
§ 14.72. Asymptotic Mathematics	245
§ 14.73. Analysis of Algorithms	245
§ 14.73.1. Sorting	245
§ 14.73.1.1. Counting Sort	245
§ 14.73.1.2. Radix Sort	245
§ 14.73.1.3. Survey of Sorting Algorithms	245
§ 14.74. RAM Model and Asymptotic Analysis	245
§ 14.74.1. What Big O doesn’t want you to know?	245
§ 14.75. An Informal Survey of Multiplication Algorithms	245
§ 14.76. Advanced Data Structures	245
§ 14.77. Datatypes (One last time)	245
§ 14.78. Stacks and Queues	245
§ 14.78.1. Stack	245
§ 14.78.2. Queue	245
§ 14.78.2.1. Amortization	246
§ 14.78.2.2. The True Queue	246
§ 14.79. Binary Search Tree	246
§ 14.80. Sets and Maps	246
§ 14.81. Exercise	246
§ 14.82. Type Classes	246

§ 14.83. Basics of Typeclasses	246
§ 14.83.1. Vector Spaces	246
§ 14.83.2. Other Typeclasses	246
§ 14.83.3. General Definition	246
§ 14.84. The Point of Typeclasses	246
§ 14.84.1. The Point of Vector Spaces	246
§ 14.84.2. The Point of any Class	246
§ 14.84.2.1. Types	246
§ 14.85. Induced Instances	246
§ 14.86. The <code>deriving</code> Keyword	247
§ 14.87. Superclasses and Subclasses	247
§ 14.88. Laws	247
§ 14.89. Monads	247
§ 14.90. Appendix	247

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 π
- 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 \doteq **mathematical value**,

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

§1.4. Variables

\doteq **mathematical variable**

A **mathematical variable** is a symbol or chain of symbols meant to represent an arbitrary element from a set of \doteq **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 \doteq **mathematical variable** as it isn't any one specific \doteq **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 \doteq **mathematical variable** as it isn't any one specific \doteq **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 \Leftarrow **mathematical variable** as it isn't any one specific \Leftarrow **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.

\Leftarrow 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 \Leftarrow **mathematical value**, or
 - a \Leftarrow **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**.

\Leftarrow 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 \Leftarrow **checking whether mathematical expression is well-formed**.

Let us use \Leftarrow **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, c, 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 middle¹, and a right-hand side.

A few examples -

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

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

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 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”.

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

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

,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 -

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

For example, consider the following definition -

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

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 -

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

then all the statements $\varphi_0, \varphi_1, \varphi_2, \varphi_3, \dots$ 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} 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 \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 φ_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 ≤ 0 is 0, and $F(0) := 1$, so the definition terminates immediately.

- $\langle\langle \text{For each } n > 0, \text{ if } \varphi_{n-1} \text{ is true, then } \varphi_n \text{ is also true.} \rangle\rangle$

Assume that φ_{n-1} is true.

Let m be an arbitrary natural number which is $\leq n$.

- $\langle\langle \text{Case 1 } (m \leq 1) \rangle\rangle$

$F(m) := 1$, so the definition terminates immediately.

- $\langle\langle \text{Case 2 } (m > 1) \rangle\rangle$

$F(m) := F(m - 1) + F(m - 2)$,

and since $m - 1$ and $m - 2$ are both $\leq n - 1$,

φ_{n-1} tells us that both $F(m - 1)$ and $F(m - 2)$ must terminate.

Thus $F(m) := F(m - 1) + F(m - 2)$ must also terminate.

Hence φ_n 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 operator²

≡ 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

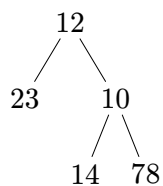
Trees are a way to structure a collection of objects.

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.

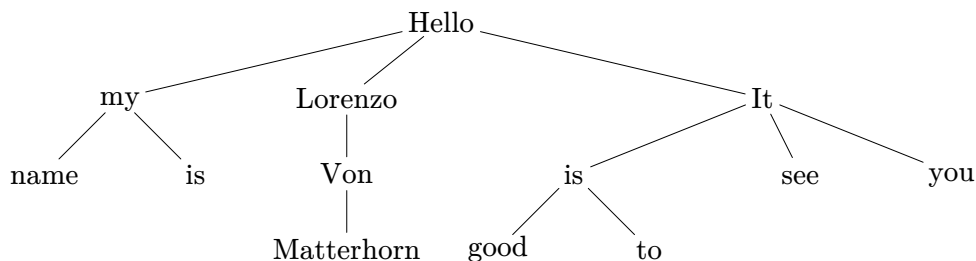
§1.8.1. Examples of Trees

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

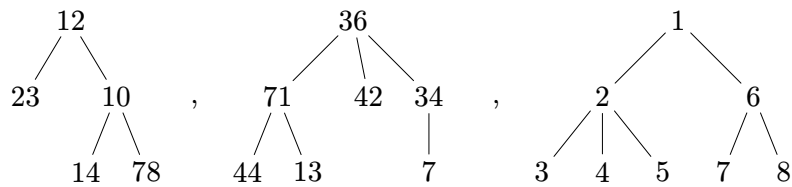
If we have an object -

89

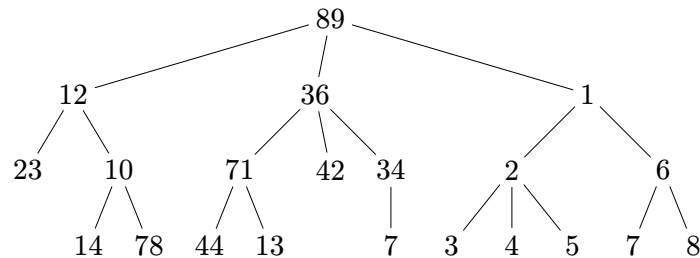
and a few trees -

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 -



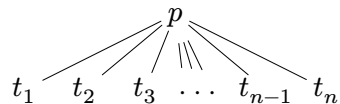
In general, if we have an object

$$p$$

and a bunch of trees

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

, 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.

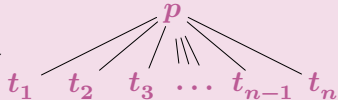
≡ checking whether object is tree

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**

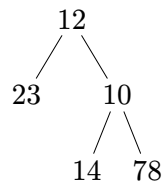
Failing that,

- 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

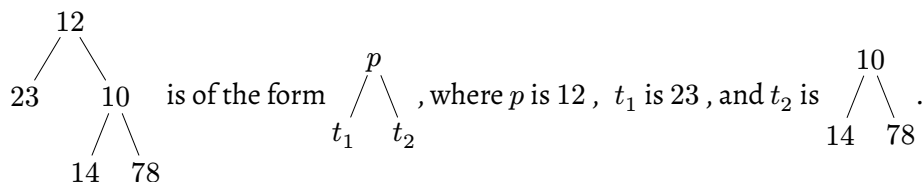
Given a set S , a **mathematical object** is said to be a **tree over S** if and only if it passes the formal checking procedure defined in ≡ checking whether object is tree.

Let us use this definition to check whether



is a tree over the natural numbers.

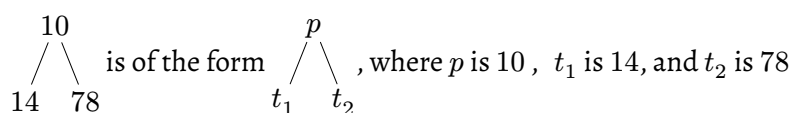
Let's start -



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

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

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



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

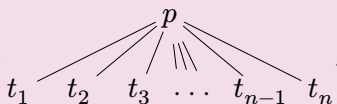
In order to prove things about trees, we have a version of the \equiv principle of mathematical induction for trees -

\equiv structural induction for trees

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

If we can prove the following two statements -

- For each $s \in S$, φ_s is true

- For each tree T of the form ,

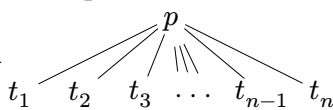
if φ_{t_1} , φ_{t_2} , φ_{t_3} , ..., $\varphi_{t_{n-1}}$ and φ_{t_n} are all true,
then φ_T is also true.

then φ_t is true for all trees t over S .

§1.8.5. Structural Recursion

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

From the definition of \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 φ_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 terminating terms always terminates)

Hence φ_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.

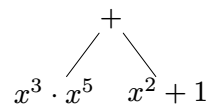
So, for dumb Haskell to do this (figure out the values of expressions, which is quite an important ability), a vital requirement is to be able to easily put expressions in the form $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$.

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

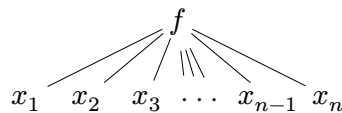
One way to make this easier to represent the expression in the form of a tree -

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

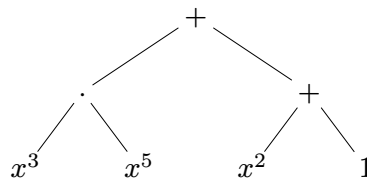


, it becomes obvious what the function is and what the inputs are to which it is applied.

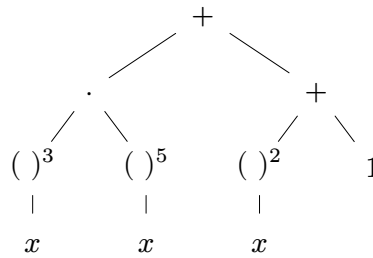
In general, we can represent the expression $f(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$ as



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

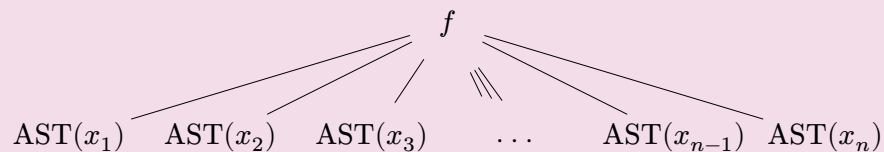
The **abstract syntax tree of a well-formed expression** is defined by applying the “function” **AST** to the expression.

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!³

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.

³Atleast 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:

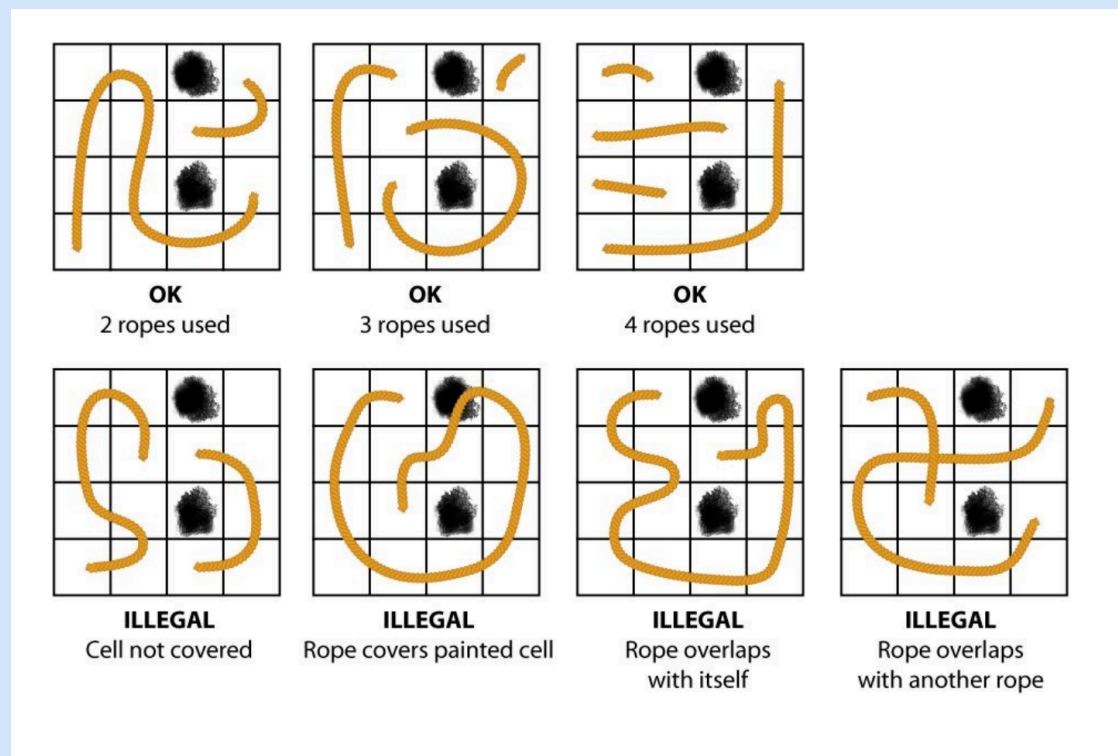


Figure out an algorithm/recipe to covering the grid using $n + 1$ ropes leagally.

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

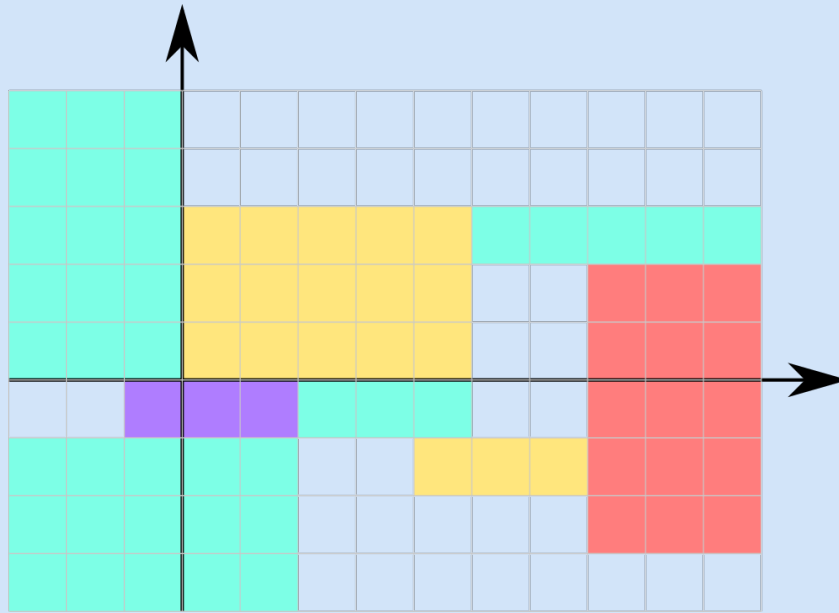
Prove that for every positive integer n , there exists an n -digit number divisible by 5^n , all of whose digits are odd.

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
 - $a \neq \text{value}$, or
 - $a \neq \text{variable}$
 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 \equiv 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 λ using infix operator as function

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

and thus can finally be shortened to the extreme

λ 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

λ 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

λ multi-input anonymous function

```
( \ x y → 1/x + 1/y )
```

λ 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
```

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.

⚠ Implementation of `abs` function

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

§3.11.1. Division, A Trilogy

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

`recip` is a function which reciprocates a given number, but it has rather interesting type signature. It is only defined on types with the `Fractional` “type-class”. This refers to a lot of things, but the most common ones are `Rational`, `Float` and `Double`. `recip`, as the name suggests, returns the reciprocal of the number taken as input. The type signature is `recip :: Fractional a => a -> a`

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

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

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

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

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

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

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

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

```

>>> 100 `div` 7
14

>>> 100 `mod` 7
2

>>> 100 `divMod` 7
(14,2)

>>> 100 `quot` 7
14

>>> 100 `rem` 7
2

>>> 100 `quotRem` 7
(14,2)

```

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

X Div vs Quot

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

```

>>> 8 `div` 3
2

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

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

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

>>> 8 `quot` 3
2

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

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

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

```

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 up to 10^{100000} to even get millisecond difference in the two).

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

ⓐ A division algorithm on positive integers by repeated subtraction

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

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

```
divideComplete :: Integer → Integer → (Integer, Integer)
divideComplete _ 0 = error "DivisionByZero"
divideComplete n d

  | d < 0      = let (q, r) = divideComplete n (-d) in
                  (-q, r)

  | n < 0      = let (q, r) = divideComplete (-n) d in
                  if r == 0 then (-q, 0) else (-q - 1, d - r)

  | otherwise = 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.

λ A naive integer exponentiation algorithm

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

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

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

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

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

λ A better exponentiation algorithm using divide and conquer

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

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

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

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

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

§3.11.3. gcd and lcm

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

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

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

>>> gcd 12 30
6

>>> lcm 12 30
60
```

We will now try to define these functions ourselves.

Let's say we want to find $g := \text{gcd}(p, q)$ 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 π upto some decimal places based on what type it is forced in.

```

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

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

```

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

§3.12.1. Binary Search

⊕ Hi-Lo game

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

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

This idea also works for slightly less direct questions:

✕ Hamburgers (Codeforces 371C)

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

At the current moment, in his pantry he has:

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

And the market prices per unit is:

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

Polycarpus's wallet has r rubles.

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

Write function

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

Examples

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

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

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

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

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

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

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

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

Here ia a similar exercise for your practice.

X House of Cards (Codeforces 471C)

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

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

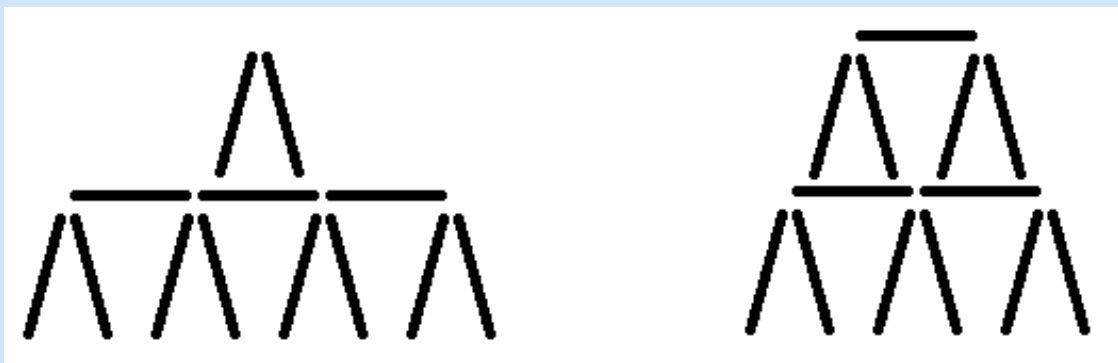
The height of the house is the number of floors in it.

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

Examples

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

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



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

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

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

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

⚠ Square root by binary search

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

✖ Cube Root

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

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

```
sqrt = bsSqrt 0.00001
```

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

§3.12.2. Taylor Series

We know that $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$. 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 1×1 squares and one 2×2 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 Baltik (COMPFEST 13, Codeforces 1575K)

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

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

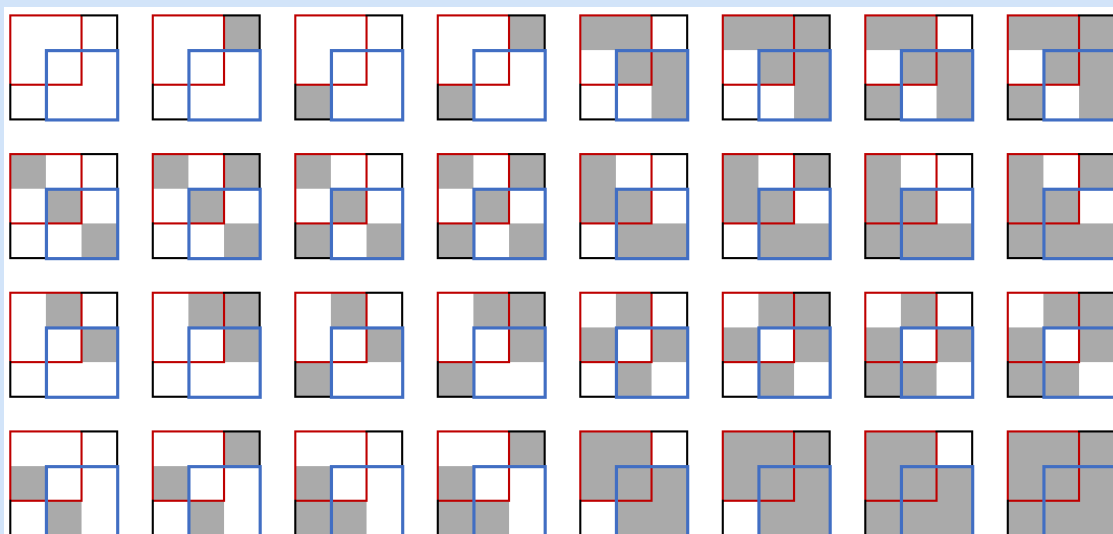
`(Int, Int) → Int → (Int, Int) → (Int, Int) → (Int, Int) → Integer` to count the number of possible batik color combinations, such that the subrectangles $((a_x, a_y), (a_x + r - 1, a_y + c - 1))$ and $((b_x, b_y), (b_x + r - 1, b_y + c - 1))$ have the same pattern.

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).

Types as Sets

§4.1. Sets

≡ set

A *set* is a *well-defined collection of “things”*.

These “things” can be values, objects, or other sets.

For any given set, the “things” it contains are called its *elements*.

Some basic kinds of sets are -

- ≡ empty set

The *empty set* is the *set that contains no elements* or equivalently, $\{\}$.

- ≡ singleton set

A *singleton set* is a *set that contains exactly one element*, such as $\{34\}$, $\{\triangle\}$, the set of natural numbers strictly between 1 and 3, etc.

We might have encountered some mathematical sets before, such as the set of real numbers \mathbb{R} or the set of natural numbers \mathbb{N} , or even a set following the rules of vectors (a vector space).

We might have encountered sets as data structures acting as an unordered collection of objects or values, such as Python sets - `set([])`, $\{1, 2, 3\}$, etc.

Note that sets can be finite ($\{12, 1, \circ, \vec{x}\}$), as well as infinite (\mathbb{N}).

A fundamental keyword on sets is “ \in ”, or “belongs”.

≡ belongs

Given a value x and a set S ,

$x \in S$ is a *claim* that *x is an element of S* ,

Other common operations include -

≡ union

$A \cup B$ is the *set containing all those x such that either $x \in A$ or $x \in B$* .

≡ intersection

$A \cap B$ is the *set containing all those x such that $x \in A$ and $x \in B$* .

≡ cartesian product

$A \times B$ is the *set containing all ordered pairs (a, b) such that $a \in A$ and $b \in B$* .

So,

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

⊃ set exponent

B^A is the *set of all functions with domain A and co-domain B* ,
or equivalently, the *set of all functions f such that $f : A \rightarrow B$* ,
or equivalently, the *set of all functions from A to B* .

✕ size of exponent set

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

§ 4.2. Types

We have encountered a few types in the previous chapter, such as `Bool`, `Integer` and `Char`. For our limited purposes, we can think about each such type as the set of all values of that type.

For example,

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

If this analogy were to extend further, we might expect to see analogues of the basic kinds of sets and the common set operations for types, which we can see in the following -

§ 4.2.1. `::` is analogous to \in or \ni belongs

Whenever we want to claim a value `x` is of type `T`, we can use the `::` keyword, in a similar fashion to \in , i.e., we can say `x :: T` in place of $x \in T$.

In programming terms, this is known as declaring the variable `x`.

For example,

- `λ declaration of x`

```
x :: Integer
x = 42
```

This reads - “Let $x \in \mathbb{Z}$. Take the value of x to be 42.”

- `λ declaration of y`

```
y :: Bool
y = xor True False
```

This reads - “Let $y \in \{\text{False}, \text{True}\}$. Take the value of y to be the \oplus of `True` and `False`.”

✕ declaring a variable

Declare a variable of type `Char`.

§ 4.2.2. `A → B` is analogous to B^A or \ni set exponent

As B^A contains all functions from A to B ,

so is each function `f` defined to take an input of type `A` and output of type `B` satisfy `f :: A → B`.

For example -

- **function**

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

- **another function**

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

- **basic function definition**

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

- **difference between declaration and function definition**

What are the differences between declaring a variable and defining a function?

§4.2.3. `(A , B)` is analogous to $A \times B$ or \equiv cartesian product

As $A \times B$ contains all pairs (a, b) such that $a \in A$ and $b \in B$,
so is every pair `(a,b)` of type `(A,B)` if `x` is of type `A` and `b` is of type `B`.

For example, if I ask GHCi to tell me the type of `(True, 'c')`, then it would tell me that the value's type is `(Bool, Char)` -

- **type of a pair**

```
>>> :type (True, 'c')
(True, 'c') :: (Bool, Char)
```

This reads - "GHCi, what is the type of `(True, 'c')`?

Answer : the type of `(True, 'c')` is `(Bool, Char)`."

If we have a type `X` with elements `x1`, `x2`, and `x3`, and another type `Y` with elements `y1` and `y2`, we can use the author-defined function `listOfAllElements` to obtain a list of all elements of certain types -

- **elements of a product type**

```
>>> listOfAllElements :: [X]
[x1,x2,x3]

>>> listOfAllElements :: [Y]
[y1,y2]

>>> listOfAllElements :: [(X,Y)]
[(x1,y1),(x1,y2),(x2,y1),(x2,y2),(x3,y1),(x3,y2)]

>>> listOfAllElements :: [(Char,Bool)]
[( '\NUL', False ), ( '\NUL', True ), ( '\SOH', False ), ( '\SOH', True ), . . . ]
```

There are two fundamental inbuilt operations from a product type -

A function to get the first component of a pair -

- **first component of a pair**

```
fst (a,b) = a
```

and a similar function to get the second component -

λ second component of a pair

```
snd (a,b) = b
```

We can define our own functions from a product type using these -

λ function from a product type

```
xorOnPair :: ( Bool , Bool ) → Bool
xorOnPair pair = ( fst pair ) ≠ ( snd pair )
```

or even by pattern matching the pair -

λ another function from a product type

```
xorOnPair' :: ( Bool , Bool ) → Bool
xorOnPair' ( a , b ) = a ≠ b
```

Also, we can define our functions to a product type -

For example, consider the useful inbuilt function `divMod`, which divides a number by another, and returns both the quotient and the remainder as a pair. Its definition is equivalent to the following -

λ function to a product type

```
divMod :: Integer → Integer → ( Integer , Integer )
divMod n m = ( n `div` m , n `mod` m )
```

x size of a product type

If a type `T` has n elements, and type `T'` has m elements, then how many elements does `(T.T')` have?

§4.2.4. `()` is analogous to $\{ \}$ singleton set

`()`, pronounced Unit, is a type that contains exactly one element.

That unique element is `()`.

So, it means that `() :: ()`, which might appear a bit confusing.

The `()` on the left of `::` is just a simple value, like `1` or `'a'`.

The `()` on the right of `::` is a type, like `Integer` or `Char`.

This value `()` is the only value whose type is `()`.

On the other hand, other types might have multiple values of that type. (such as `Integer`, where both `1` and `2` have type `Integer`.)

We can even check this using `listOfAllElements` -

λ elements of unit type

```
>>> listOfAllElements :: [()]
[()]
```

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

x function to unit

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

x function from unit

Define a function of type $() \rightarrow \text{Bool}$.

§4.2.5. No \ni intersection of Types

We now need to discuss an important distinction between sets and types. While two different sets can have elements in common, like how both \mathbb{R} and \mathbb{N} have the element 10 in common, on the other hand, two different types T_1 and T_2 cannot have any common elements.

For example, the types `Int` and `Integer` have no elements in common. We might think that they have the element `10` in common, however, the internal structures of `10 :: Int` and `10 :: Integer` are very different, and thus the two `10`s are quite different.

Thus, the intersection of two different types will always be empty and doesn't make much sense anyway. Therefore, no intersection operation is defined for types.

§4.2.6. No \ni union of Types

Suppose the type $T_1 \cup T_2$ were an actual type. It would have elements in common with the type T_1 . As discussed just previously, this is undesirable and thus disallowed.

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

x subtype

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

§4.2.7. Disjoint Union of Sets

\ni disjoint union

$A \sqcup B$ is defined to be $(\{0\} \times A) \cup (\{1\} \times B)$, or equivalently, *the set of all pairs either of the form $(0, a)$ such that $a \in A$, or of the form $(1, b)$ such that $b \in B$.*

So,

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

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

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

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

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

§4.2.8. `Either A B` is analogous to $A \sqcup B$ or \ni disjoint union

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

Recall that in a \oplus **disjoint union**, each element has to be

- of the form $(0, a)$, where $a \in A$, and A is the set to the left of the \sqcup symbol,
- or they can be of the form $(1, b)$, where $b \in B$, and B is the set to the right of the \sqcup symbol.

Similarly, in **Either A B**, each element has to be

- of the form **Left a**, where $a :: A$
- or of the form **Right b**, where $b :: B$

If we have a type **X** with elements **X1**, **X2**, and **X3**, and another type **Y** with elements **Y1** and **Y2**, we can use the author-defined function **listOfAllElements** to obtain a list of all elements of certain types -

elements of an either type

```
>>> listOfAllElements :: [X]
[X1,X2,X3]

>>> listOfAllElements :: [Y]
[Y1,Y2]

>>> listOfAllElements :: [Either X Y]
[Left X1,Left X2,Left X3,Right Y1,Right Y2]

>>> listOfAllElements :: [Either Bool Char]
[Left False,Left True,Right '\NUL',Right '\SOH',Right '\STX', . . . ]
```

We can define functions to an **Either** type.

Consider the following problem : We have to make a function that provides feedback on a quiz. We are given the marks obtained by a student in the quiz marked out of 10 total marks. If the marks obtained are less than 3, return **'F'**, otherwise return the marks as a percentage -

function to an either type

```
feedback :: Integer -> Either Char Integer
--
-- Left ~ Char,Integer ~ Right
feedback n
  | n < 3      = Left 'F'
  | otherwise = Right ( 10 * n ) -- multiply by 10 to get percentage
```

This reads - “

Let **feedback** be a function that takes an **Integer** as input and returns **Either** a **Char** or an **Integer**.

As **Char** and **Integer** occurs on the left and right of each other in the expression **Either Char Integer**, thus **Char** and **Integer** will henceforth be referred to as **Left** and **Right** respectively.

Let the input to the function **feedback** be **n**.

If **n<3**, then we return **'F'**. To denote that **'F'** is a **Char**, we will tag **'F'** as **Left**. (remember that **Left** refers to **Char**!)

otherwise, we will multiply `n` by `10` to get the percentage out of 100 (as the actual quiz is marked out of 10). To denote that the output `10*n` is an `Integer`, we will tag it with the word `Right`. (remember that `Right` refers to `Integer`!)

“

We can also define a function from an `Either` type.

Consider the following problem : We are given a value that is either a boolean or a character. We then have to represent this value as a number.

```
top
import Data.Char(ord)
```

λ another function from an either type

```
representAsNumber :: Either Bool Char → Int
--                Left ~ Bool, Char ~ Right
representAsNumber ( Left  bool ) = if bool then 1 else 0
representAsNumber ( Right char ) = ord char
```

This reads - “

Let `representAsNumber` be a function that takes either a `Bool` or a `Char` as input and returns an `Int`.

As `Bool` and `Char` occurs on the left and right of each other in the expression `Either Bool Char`, thus `Bool` and `Char` will henceforth be referred to as `Left` and `Right` respectively.

If the input to `representAsNumber` is of the form `Left bool`, we know that `bool` must have type `Bool` (as `Left` refers to `Bool`). So if the `bool` is `True`, we will represent it as `1`, else if it is `False`, we will represent it as `0`.

If the input to `representAsNumber` is of the form `Right char`, we know that `char` must have type `Char` (as `Right` refers to `Char`). So we will represent `char` as `ord char`.

“

We might make things clearer if we use a deeper level of pattern matching, like in the following function (which is equivalent to the last one).

λ another function from an either type

```
representAsNumber' :: Either Bool Char → Int
representAsNumber' ( Left  False ) = 0
representAsNumber' ( Left  True  ) = 1
representAsNumber' ( Right char ) = ord char
```

✕ size of an either type

If a type `T` has n elements, and type `T'` has m elements, then how many elements does `Either T T'` have?

§4.2.9. The `Maybe` Type

Consider the following problem : We are asked make a function `reciprocal` that reciprocates a rational number, i.e., $(x \mapsto \frac{1}{x}) : \mathbb{Q} \rightarrow \mathbb{Q}$.

Sounds simple enough! Let's see -

`λ naive reciprocal`

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

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

What should be the output of `reciprocal 0`?

Unfortunately, it results in an error -

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

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

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

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

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

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

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

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

`λ reciprocal using either`

```
reciprocal :: Rational → Either () Rational
reciprocal 0 = Left ()
reciprocal x = Right (1/x)
```

There is already an inbuilt way to express this notion of `Either () Rational` in Haskell, which is the type `Maybe Rational`.

`Maybe Rational` just names it elements a bit differently compared to `Either () Rational` -

- where

`Either () Rational` has `Left ()`,

`Maybe Rational` instead has the value `Nothing`.

- where

`Either () Rational` has `Right r` (where `r` is any `Rational`),

`Maybe Rational` instead has the value `Just r`.

Which means that we can rewrite `λ reciprocal using either` using `Maybe` instead -

λ function to a maybe type

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

But we can also do this for any arbitrary type `T` in place of `Rational`. In that case -

There is already an inbuilt way to express the notion of `Either () T` in Haskell, which is the type `Maybe T`.

`Maybe T` just names it elements a bit differently compared to `Either () T` -

- where

`Either () T` has `Left ()`,

`Maybe T` instead has the value `Nothing`.

- where

`Either () T` has `Right t` (where `t` is any value of type `T`),

`Maybe T` instead has the value `Just t`.

If we have a type `X` with elements `x1`, `x2`, and `x3`, and another type `Y` with elements `y1` and `y2`, we can use the author-defined function `listOfAllElements` to obtain a list of all elements of certain types -

λ elements of a maybe type

```
>>> listOfAllElements :: [X]
[X1,X2,X3]

>>> listOfAllElements :: [Maybe X]
[Nothing,Just X1,Just X2,Just X3]

>>> listOfAllElements :: [Y]
[Y1,Y2]

>>> listOfAllElements :: [Maybe Y]
[Nothing,Just Y1,Just Y2]

>>> listOfAllElements :: [Maybe Bool]
[Nothing,Just False,Just True]

>>> listOfAllElements :: [Maybe Char]
[Nothing,Just '\NUL',Just '\SOH',Just '\STX',Just '\ETX', . . . ]
```

✕ size of a maybe type

If a type `T` has n elements, then how many elements does `Maybe T` have?

We can define functions to a `Maybe` type. For example consider the problem of making an inverse function of `reciprocal`, i.e., a function `inverseOfReciprocal` s.t.

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

as follows -

⚠ function from a maybe type

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

§4.2.10. `Void` is analogous to $\{\}$ or \emptyset empty set

The type `Void` has no elements at all.

This also means that no actual value has type `Void`.

Even though it is out-of-syllabus, an interesting exercise is to

✕ Exercise

try to define a function of type `(Bool → Void) → Void`.

§4.3. Currying

Let's try to explore some more elaborate types.

For example, let us try to find out the type of the derivative operator,

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

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

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

Specifically, we could define

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

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

And thus we obtain the type of the derivative operator as

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

or more formally,

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

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

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

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

Here, the definition could be written as

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

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

That leads us to the type -

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

We understand that these two definitions are equivalent.

So now the question is, which type do we use?

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

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

In general, that means that if a function $F : A \rightarrow (B \rightarrow C)$
takes an input from A ,
and gives as output a $B \rightarrow C$ function,
then it is equivalent to saying $F : A \times B \rightarrow C$, which would make F a function
that takes inputs of type A and B respectively,
and gives an output of type C .

5

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

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

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

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

and rephrasing it as

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

In general, if a function $F : A \times B \rightarrow C$
takes inputs of type A and B respectively,
and gives an output of type C .
then it is equivalent to saying $F : A \rightarrow (B \rightarrow C)$, which would make F a function
that takes an input from A ,
and gives as output a $B \rightarrow C$ function.

6

⁵will be proven soon

⁶will be proven soon

This rephrasing is known as “currying”.

§4.3.1. In Haskell

Again, for example,

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

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

Let’s see how it works in Haskell.

≡ currying rule

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

$$f\ x = \lambda y \rightarrow f\ x\ y$$

We know that $(+)$ is a function that takes in two `Integer`s and outputs an `Integer`.

This means that A, B, C are `Integer`, `Integer`, `Integer` respectively.

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

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

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

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

More explicitly, by the ≡ currying rule, we have that

$$(+)\ 17 = \lambda y \rightarrow (+)\ 17\ y$$

Thus, what does this function $(+)\ 17$ actually do?

Simple! It is a function that takes in any `Integer` and adds `17` to it.

So, for example,

If we define -

λ currying usage

```
test = (+) 17
```

it behaves as such -

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

Another -

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

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

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

More explicitly, by the \Rightarrow **currying rule**, we have that

```
(*) 2 = \ y → (*) 2 y
```

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

So if we define -

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

it behaves as such -

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

§4.3.2. Understanding through Associativity

§4.3.2.1. Of \rightarrow

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

This is due to the fact that as an \Rightarrow **infix binary operator**, the \rightarrow operator is \Rightarrow **right-associative**.

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

$$X \rightarrow Y \rightarrow Z$$

is actually equivalent to

$$X \rightarrow (Y \rightarrow Z)$$

And thus the \equiv **currying rule** is justified.

§4.3.2.2. Of Function Application

Let us take the \equiv **currying rule**

$$f\ x = \lambda y \rightarrow f\ x\ y$$

Applying a few transformations to both sides -

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

Thus we obtain the result that any time we write

$$f\ x\ y$$

it is actually equivalent to

$$(f\ x)\ y$$

This means that “function application” is \equiv **left-associative**. (Recall the definition of \equiv **left-associative** and see if this makes sense)

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

§4.3.2.3. Operator Currying Rule

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

⚡ operator currying rule

If we have an \div infix binary operator $\boxed{?}$, then we can assume the following due to the \div currying rule -

```
( $\boxed{?}$ ) x = \ y  $\rightarrow$  ( $\boxed{?}$ ) x y -- the normal currying rule
-- which is equivalent to
( $\boxed{?}$ ) x = \ y  $\rightarrow$  x  $\boxed{?}$  y
```

But we may further assume

```
(x $\boxed{?}$ ) = \ y  $\rightarrow$  x  $\boxed{?}$  y
```

and also

```
( $\boxed{?}$ y) = \ x  $\rightarrow$  x  $\boxed{?}$  y
```

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

For example, -

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

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

Meaning that when the \div infix binary operator \wedge is given the `Integer` input `2` in place of its second input, it outputs the function `(^2)`, of type `Integer -> Integer`.

More explicitly, by the \div operator currying rule -

```
(^2) = \ x -> x ^ 2
```

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

So if we define -

⚡ operator currying usage

```
squaring :: Integer -> Integer
squaring = (^2)
```

it will show the following behaviour -

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

For another example, we can define

```
λ another operator currying usage
cubing :: Integer → Integer
cubing = (^3)
```

which works quite similarly.

§4.3.3. Proof of the Currying Theorem

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

In general, if a function $F : A \times B \rightarrow C$ takes inputs of type A and B respectively, and gives an output of type C .

then it is equivalent to saying $F : A \rightarrow (B \rightarrow C)$, which would make F a function that takes an input from A , and gives as output a $B \rightarrow C$ function.

What we are going to prove is that

there exists a bijection

from

the set $\{F \mid F : A \times B \rightarrow C\}$

to

the set $\{G \mid G : A \rightarrow (B \rightarrow C), \text{ or equivalently, } G : A \rightarrow C^B\}$

Note that the set $\{F \mid F : A \times B \rightarrow C\}$ can be expressed as $C^{A \times B}$

and that the set $\{G \mid G : A \rightarrow C^B\}$ can be expressed as $(C^B)^A$

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

ⓧ [finite currying](#)

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

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

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

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

If we can prove that \mathcal{C} is bijective, we are done!

In order to do that, we will prove that \mathcal{C} is injective as well as a surjective.

Claim : \mathcal{C} is injective

Proof :

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

Claim : \mathcal{C} is surjective

Proof : Take an arbitrary $H \in (C^B)^A$.

In other words, take an arbitrary function $H : A \rightarrow (B \rightarrow C)$.

Define a function J as follows -

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

Now,

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

That means we have proved that

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

Therefore, by the definition of surjectivity, we have proven that \mathcal{C} is surjective.

As a result, we are done with the overall proof as well!

■

§4.4. Exercises

x Symmetric Difference

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

Prove that this is a commutative and associative operation.

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

x Set of Size

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

x Unions and Intersections

Prove: (i)

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

(ii)

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

(iii)

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

(iv)

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

(v)

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

(vi)

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

x Flavoured like Curry

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

Prove:

(i)

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

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

(iii)

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

x Eckman-Hilton Argument

≡ Unital Operator

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

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

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

Prove that $\star \equiv \cdot$.

x Associative Operators

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

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

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

X Feels Abstract

÷ Shelf

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

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

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

÷ Rack

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

- (ii) Prove that in a rack R , \triangleright distributes over \triangleleft and vice versa. That is, $\forall x, y, z \in R; x \triangleright (y \triangleleft z) = (x \triangleright y) \triangleleft (x \triangleright z)$.
- (iii) We call an operator \star idempotent if $x \star x = x$. If for a rack, \triangleright is idempotent; prove that \triangleleft is idempotent.

÷ Quandle

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

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

÷ Kei

An involutive quandle is called a Kei

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

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

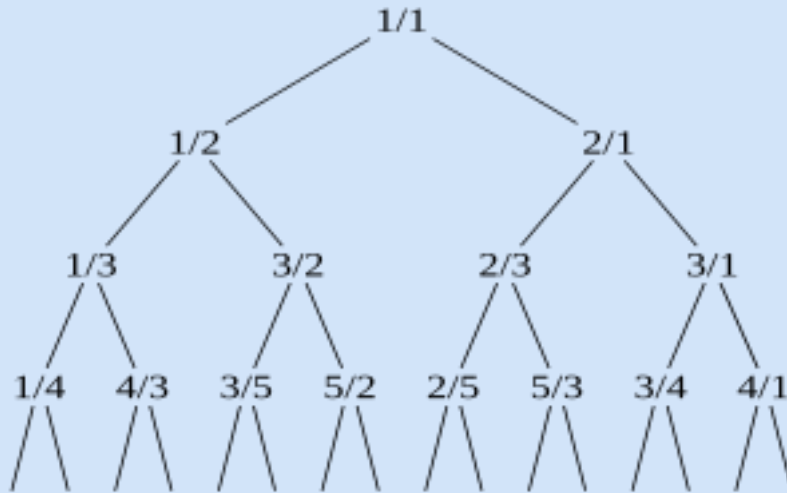
x Enumerating the Rationals

- (i) Prove that \mathbb{Z} is countable, that is there is a surjection from $\mathbb{N} \rightarrow \mathbb{Z}$.⁷
- (ii) Now write functions : `natToInt :: Integer → Integer` and `intToNat :: Integer → Integer` which takes a natural number and gives the corresponding integer and vice versa.
- (iii) Prove that $\mathbb{N} \times \mathbb{N}$ is countable, that is there is a bijection between $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$. While other arguments exist, we are big fans of enumerating in the order $(0, 0) \rightarrow (0, 1) \rightarrow (1, 0) \rightarrow (0, 2) \rightarrow (1, 1) \rightarrow (2, 0) \rightarrow \dots$. What is the pattern?
- (iv) Now write a function `intToPair :: Int → (Int, Int)` which takes an integer and gives the corresponding rational and a function `pairToInt :: (Int, Int) → Int` which takes a pair and returns the corresponding integer.

x Calkin-Wilf Tree

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

- (i) Prove that if $\frac{p}{q}$ is reduced, then $\frac{p+q}{q}$ and $\frac{p}{p+q}$ are reduced.
- (ii) Prove that starting with $\frac{1}{1}$ and making the following tree by applying the above transformation will contain every rational:



Introduction to Lists

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

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

These objects are called the elements of the list.

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

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

§5.1. Type of List

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

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

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

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

§5.2. Creating Lists

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

§5.2.1. Empty List

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

§5.2.2. Arithmetic Progression

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

λ arithmetic progression syntax

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

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

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

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

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

λ non-number arithmetic progressions

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

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

§5.3. Functions on Lists

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

§5.4. List Comprehension

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

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

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

Haskell lets us do the same with lists -

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

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

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

,which can be expressed in haskell as

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

§5.4.1. Cons or `(:)`

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

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

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

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

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

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

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

So, how can we use it for pattern matching?

▶ pattern matching lists

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

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

§5.5. Length

One of the most basic questions we could ask about lists is the number of elements they contain.

The `length` function gives us that answers, counting repetitions as separate.

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

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

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

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

Ans we can use pattern matching to define it -

▶ length of list

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

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

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

§5.5.1. Concatenate or `(++)`

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

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

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

Again, we can define it by using pattern matching

```
λ concatenation of lists
[]      ++ ys = ys
(x:xs) ++ ys = x : ( xs ++ ys )
```

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

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

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

§5.5.2. Head and Tail

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

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

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

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

And it can be defined using pattern-matching -

```
λ head of list
head (x:xs) = x
```

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

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

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

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


And it can be defined using pattern-matching -

```
λ tail of list
tail (x:xs) = xs
```

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

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

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

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

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

λ *function to a maybe type*, which is

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

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

Let's test this in GHCi -

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

And the definition -

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

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

§5.5.3. Take and Drop

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

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

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

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

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

And the definition -

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

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

If we want to take n elements, then we can take the first element and then the first n-1 elements from the rest.

But why the last line of the definition? “The last line of the function may look strange, but -

x Exercise

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

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

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

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

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

And the definition -

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

x Exercise

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

The splitAt function combines these two functionalities by returning both answers in a pair.

That is; `splitAt n l == (take n l , drop n l)`

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

§5.5.4. (!!)

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

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

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

Error!

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

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

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

x Exercise

Provide a definition for the `!!` operator.

§5.5.5. List \rightarrow Bool

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

§5.5.5.1. Elem

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

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

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

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

And the definition -

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

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

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

§5.5.5.2. Generalized Logical Operators

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

`and`

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

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

A few examples -

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

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

>>> and []
True

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

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

Let's generalize `||` as well.

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

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

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

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

>>> or []
False

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

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

x base cases of list-ary logical operators

Try to justify why the definitions `and [] = True` and `or [] = False` are required.

§5.6. Strings

A string is how we represent text (like English sentences and words) in programming.

Like many modern programming languages, Haskell defines a string to be just a list of characters.

In fact, the type `String` is just a way to refer to the actual type `[Char]`.

So, if we want write the text “hello there!”, we can write it in GHCi as `['h','e','l','l','o',' ','t','h','e','r','e','!']`.

Let's test it out -

```
>>> ['h','e','l','l','o',' ','t','h','e','r','e','!']
"hello there!"
```

But we see GHCi replies with something much simpler - `"hello there!"`

This simplified form is called syntactic sugar. It allows us to read and write strings in a simple form without having to write their actual verbose syntax each time.

So, we can write -

```
>>> "hello there!"
"hello there!"

>>> :type +d "hello there!"
"hello there!" :: String
```

The type `String` is just a way to refer to the actual type `[Char]`.

And since strings are just lists, all the list functions apply to strings as well.

```
>>> 'h' : "ello there!"
"hello there!"

>>> "hello " ++ "there!"
"hello there!"

>>> head "hello there!"
'h'

>>> tail "hello there!"
"ello there!"

>>> take 5 "hello there!"
"hello"

>>> drop 5 "hello there!"
" there!"

>>> elem 'e' "hello there!"
True

>>> elem 'w' "hello there!"
False

>>> "hello there!" !! 7
'h'

>>> "hello there!" !! 6
't'
```

But there are some special functions just for strings -

`words` breaks up a string into a list of the words in it.

```
>>> words "hello there!"
["hello","there!"]
```

And `unwords` combines the words back into a single string.

```
>>> unwords ["hello", "there!"]
"hello there!"
```

`lines` breaks up a string into a list of the lines in it.

```
>>> lines "hello there!\nI am coding... "
["hello there!", "I am coding... "]
```

Ans `unlines` combines the lines back into a single string.

```
>>> unlines ["hello there!", "I am coding... "]
"hello there!\nI am coding... \n"
```

§5.7. Structural Induction for Lists

Suppose we want to prove some fact about lists.

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

\equiv **structural induction for lists**

Suppose for each list `l` of type `[T]`, we have a statement φ_l . If we can prove the following two statements -

- $\varphi []$
- For each list of the form `(x:xs)`, if φ_{xs} is true, then $\varphi_{(x:xs)}$ is also true.

then φ_l for all finite lists `l`.

Let us use this principle to prove that

Theorem The definition of `length` terminates on all finite lists.

Proof Let φ_l be the statement

The definition of `length l` terminates.

To use \equiv **structural induction for lists**, we need to prove -

- $\langle\langle \varphi [] \rangle\rangle$
The definition of `length []` directly gives `0`.
- $\langle\langle \text{For each list } (x:xs), \text{ if } \varphi_{xs}, \text{ then } \varphi_{(x:xs)} \text{ also.} \rangle\rangle$

Assume φ_{xs} is true.

The definition for `length (x:xs)` is `1 + length xs`.

By φ_{xs} , we know that `length xs` will finally give return some number `n`.

Therefore `1 + length xs` reduces to `1 + n`.

And `1 + n` obviously terminates. ■

§5.8. Sorting

≡ sorted list

A list is said to be **sorted**
if and only if
its elements appear in ascending order of their values.

OR EQUIVALENTLY

A list $[x_1, x_2, x_3, \dots, x_{n-1}, x_n]$ is said to be **sorted**
if and only if

$$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_{n-1} \leq x_n$$

OR EQUIVALENTLY

A list $[x_1, x_2, x_3, \dots, x_{n-1}, x_n]$ is said to be **sorted**
if and only if

$$(x_1 \leq x_2) \ \&\& \ (x_2 \leq x_3) \ \&\& \ (x_3 \leq x_4) \ \&\& \ \dots \ \&\& \ (x_{n-1} \leq x_n)$$

Here are a few examples of ≡ sorted lists -

```
[1, 2, 3, 4, 5]
[0, 10, 20, 30, 40]
[-10, -5, 0, 5, 10]
[2, 3, 5, 7, 11, 13, 17]
[100, 200, 300, 400, 500]
[-100, -50, -10, -1, 0, 1, 10]
[1, 1, 2, 3, 5, 8, 13]
```

and here are few which are NOT ≡ sorted lists -

```
[5, 2, 4, 1, 3]
[30, 10, 40, 0, 20]
[10, -5, 0, -10, 5]
[11, 2, 17, 5, 13, 3, 7]
[500, 100, 300, 200, 400]
[10, -1, -100, 0, -50, -10, 1]
[8, 1, 13, 5, 3, 1, 2]
```

Let's write a function that takes a list of and answers whether it is a ≡ sorted list or not -


```
isSorted [] = True
isSorted [x] = True
isSorted (x:x':xs) = ( x ≤ x' ) && ( isSorted (x':xs) )
```

This reads - “A list which contains nothing is a \doteq sorted list.

A list containing exactly one element is also a \doteq sorted list.

If the first two elements of the list are x and x' and the rest of the elements form a list xs , then the list is sorted if and only if

$x \leq x'$ AND the list $x':xs$ (i.e., the λ tail of list $x:x':xs$, the given input) is sorted. “

Now we introduce an infamous problem in computer science, “sorting”!

\doteq sorting

Sorting is the act of taking a given list and rearranging the contained elements so that it becomes a \doteq sorted list.

In Haskell, we do this using the `sort` function.

```
>>> sort [5, 2, 4, 1, 3]
[1,2,3,4,5]

>>> sort [30, 10, 40, 0, 20]
[0,10,20,30,40]

>>> sort [10, -5, 0, -10, 5]
[-10,-5,0,5,10]

>>> sort [11, 2, 17, 5, 13, 3, 7]
[2,3,5,7,11,13,17]

>>> sort [500, 100, 300, 200, 400]
[100,200,300,400,500]

>>> sort [10, -1, -100, 0, -50, -10, 1]
[-100,-50,-10,-1,0,1,10]

>>> sort [8, 1, 13, 5, 3, 1, 2]
[1,1,2,3,5,8,13]
```

Let us see whether we can define the `sort` function.

Well, it is obvious that

```
sort [] = []
```

So we are left with defining `sort (x:xs)`.

In the style of recursive definitions, we can assume that we already have `sort xs` computed.

i.e., let us define

```
sortedTail = sort xs
```

and we can henceforth use `sortedTail` to refer to `sort xs`.

Now, `sortedTail`, being `sort xs`, contains all the elements of `xs`, rearranged in ascending order. But it doesn't contain `x`.

If we were able to include `x` in `sortedTail` without disturbing this ascending order, we would be done!

So let's do that -

First we take those elements of `sortedTail` which should appear before `x` in the ascending order. (i.e., the elements `< x`)

```
[e | e ← sortedTail , e < x]
```

Then we follow that with `x` itself.

```
[e | e ← sortedTail , e < x] ++ [x]
```

And then we add the elements of `sortedTail` that should appear after `x` in the ascending order. (i.e., the elements `≥ x`)

```
[e | e ← sortedTail , e < x] ++ [x] ++ [e | e ← sortedTail , e ≥ x]
```

And thus we obtain a list containing `x` as well as all the elements of `sortedTail`, arranged in ascending order, i.e., `sort (x:xs)`

Putting it all together, we can write a definition for `sort` as follows -

```
λ sort
sort []      = []
sort (x:xs) = let sortedTail = sort xs in
  [e | e ← sortedTail , e < x] ++ [x] ++ [e | e ← sortedTail , e ≥ x]
```

Let's see an example computation -

```

sort [5, 1, 13, 8, 3, 1, 2]
== let sortedTail = sort [1, 13, 8, 3, 1, 2] in
   [e | e ← sortedTail, e < 5] ++ [5] ++ [e | e ← sortedTail, e ≥ 5]
== let sortedTail = [1,1,2,3,8,13] in
   [e | e ← sortedTail, e < 5] ++ [5] ++ [e | e ← sortedTail, e ≥ 5]
== let sortedTail = [1,1,2,3,8,13] in
   [1,1,2,3] ++ [5] ++ [e | e ← sortedTail, e ≥ 5]
== let sortedTail = [1,1,2,3,8,13] in
   [1,1,2,3] ++ [5] ++ [8,13]
== [1,1,2,3] ++ [5] ++ [8,13]
== [1,1,2,3,5,8,13]

```

§5.9. Optimization

Suppose we want to reverse the the order of elements in a list.

For example, transforming the list `[5,8,3,2,0]` into `[0,2,3,8,5]`.

So how do we define the function `reverse`?

An obvious definition is -

λ `naive reverse`

```

reverse []      = []
reverse (x:xs) = ( reverse xs ) ++ [x]

```

But this is not “optimal”.

What does this mean? Let’s see -

Let’s apply the definitions of `reverse` and `(++)` to see how `reverse [5,8,3]` is computed -

```

reverse [5,8,3] == ( reverse [8,3] ) ++ [5]
               == ( ( reverse [3] ) ++ [8] ) ++ [5]
               == ( ( ( reverse [] ) ++ [3] ) ++ [8] ) ++ [5]
               == ( ( [] ++ [3] ) ++      [8] ) ++      [5]
               == (      [3] ++      [8] ) ++      [5]
               == (      3  : ( [] ++ [8] ) ) ++      [5]
               == (      3  :      [8] ) ++      [5]
               == (      3  : (      [8] ++      [5] ) )
               ==      3  : (      8  : ( [] ++ [5] ) )
               ==      3  : (      8  :      [5] )

-- which finally is
      [3,8,5]

```

So we see that this takes 10 steps of computation.

Let us take an alternative definition of `reverse` -

λ **optimized reverse**

```

reverse l = help [] l where
  help xs (y:ys) = help (y:xs) ys
  help xs []     = xs

```

Let us how this one is computed step by step -

```

reverse [5,8,3] == help [] [5,8,3]
               == help [5] [8,3]
               == help [8,5] [3]
               == help [3,8,5] []
               == [3,8,5]

```

So we see this computation takes only 5 steps, as compared to 10 from last time.

So, in some way, the second definition is better as it requires much less steps.

We can comment on something similar for `splitAt`

λ **naive splitAt**

```

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

```

λ **optimized splitAt**

```

splitAt n [] = []
splitAt n (x:xs) = ( x:ys , zs ) where
  (ys,zs) = splitAt (n-1) xs

```

X Exercise

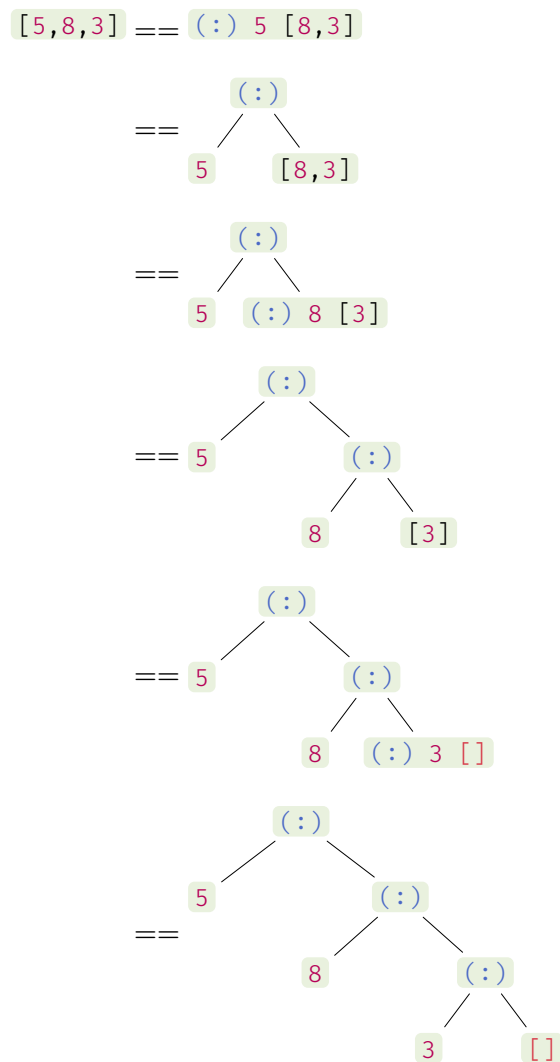
- (1) Prove that the two definitions are equivalent using \equiv **structural induction for lists**.
- (2) See which definition takes more steps to compute `splitAt 2 [5,8,3]`

§5.10. Lists as Syntax Trees

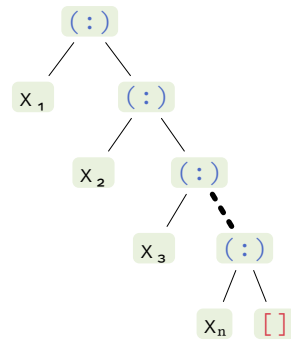
Recall \equiv **abstract syntax tree**.

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

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



In fact any list $[x_1, x_2, x_3, \dots, x_n]$ can be represented as



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

§5.11. Dark Magic

We can use our arithmetic progression notation to generate infinite arithmetic progressions.

```
>>> [0..]
[0,1,2,3,4,5,6,7,8,9, ... ]

>>> [2,5..]
[2,5,8,11,14,17,20,23,26,29, ... ]
```

We can define infinite lists like -

a list of infinitely many 0s -

```
zeroes = 0 : zeroes
```

```
>>> zeroes
[0,0,0,0,0,0,0,0,0,0, ... ]
```

the list of all natural numbers -

```
naturals = l 0 where l n = n : l (n+1)
```

```
>>> naturals
[0,1,2,3,4,5,6,7,8,9, ... ]
```

and the list of all fibonacci numbers -

```
fibs = l 0 1 where l a b = a : l b (a+b)
```

```
>>> fibs
[0,1,1,2,3,5,8,13,21,34, ... ]
```

Since we obviously cannot view the entirety of an infinite list, it is advisable to use `take` to view an initial section of the list, rather than the whole thing.

§5.11.1. Exercises

x **Balloons**

In an ICPC contest, balloons are distributed as follows:

- Whenever a team solves a problem, that team gets a balloon.
- The first team to solve a problem gets an additional balloon.

A contest has 26 problems, labelled A, B, \dots, Z . You are given the order of solved problems in the contest, denoted as a string s , where the i -th character indicates that the problem s_i has been solved by some team. No team will solve the same problem twice.

Write a function `balloons :: String → Int` to determine the total number of balloons used in the contest. Note that some problems may be solved by none of the teams.

Example :

```
balloons "ABA" = 5
balloons "A" = 2
balloons "ORZ" = 6
balloons "BAAAA" = 7
balloons "BAAAA" = 7
balloons "BKPT" = 8
balloons "BKPT" = 8
balloons "HASKELL" = 13
```

x **Neq Array (INOI 2025 P1)**

Given a list A of length N , we call a list of integers B of length N such that:

- All elements of B are positive, ie $\forall 1 \leq i \leq N, B_i > 0$
- B is non-decreasing, ie $B_1 \leq B_2 \leq \dots \leq B_N$
- $\forall 1 \leq i \leq N, B_i = A_i$

Let $\text{neq}(A)$ denote the minimum possible value of the last element of B for a valid array B .

Write a function `neq :: [Int] → Int` that takes a list A and returns the $\text{neq}(A)$.

Example :

```
neq [2,1] = 2
neq [1,2,3,4] = 5
neq [2,1,1,3,2,1] = 3
```

X Kratki (COCI 2014)

Given two integers N and K , write a function `krat :: Int → Int → Maybe [Int]` which constructs a permutation of numbers from 1 to N such that the length of its longest monotone subsequence (either ascending or descending) is exactly K or declare that the following is not possible.

A monotone subsequence is a subsequence where elements are either in non-decreasing order (ascending) or non-increasing order (descending).

Example:

```
krat 4 3 = Just [1,4,2,3]
krat 5 1 = Nothing
krat 5 5 = Just [1,2,3,4,5]
```

For example 1: The permutation (1, 4, 2, 3) has longest ascending subsequence (1, 2, 3) of length 3, and no longer monotone subsequence exists. For example 2: It's impossible to create a permutation of 5 distinct numbers with longest monotone subsequence of length 1. For example 3: The permutation (1, 2, 3, 4, 5) itself is the longest monotone subsequence of length 5.

Putnik (COCI 2013)

Chances are that you have probably already heard of the travelling salesman problem. If you have, then you are aware that it is an NP-hard problem because it lacks an efficient solution. Well, this task is an uncommon version of the famous problem! Its uncommonness derives from the fact that this version is, actually, solvable.

Our vacationing mathematician is on a mission to visit N cities, each exactly once. The cities are represented by numbers $1, 2, \dots, N$. What we know is the direct flight duration between each pair of cities. The mathematician, being the efficient woman that she is, wants to modify the city visiting sequence so that the total flight duration is the minimum possible.

Alas, all is not so simple. In addition, the mathematician has a peculiar condition regarding the sequence. For each city labeled K must apply: either all cities with labels smaller than K have been visited before the city labeled K or they will all be visited after the city labeled K . In other words, the situation when one of such cities is visited before, and the other after is not allowed.

Assist the vacationing mathematician in her ambitious mission and write a function `time :: [[Int]] → Int` to calculate the minimum total flight duration needed in order to travel to all the cities, starting from whichever and ending in whichever city, visiting every city exactly once, so that her peculiar request is fulfilled. Example :

```
time [
  [0,5,2],
  [5,0,4],
  [2,4,0]] = 7

time [
  [0,15,7,8],
  [15,0,16,9],
  [7,16,0,12],
  [8,9,12,0]] = 31
]
```

In the first example: the optimal sequence is 2, 1, 3 or 3, 1, 2. The sequence 1, 3, 2 is even more favourable, but it does not fulfill the condition. In the second example: the sequence is either 3, 1, 2, 4 or 4, 2, 1, 3.

Look and Say

Look-and-say sequences are generated iteratively, using the previous value as input for the next step. For each step, take the previous value, and replace each run of digits (like 111) with the number of digits (3) followed by the digit itself (1).

For example:

- 1 becomes 11 (1 copy of digit 1).
- 11 becomes 21 (2 copies of digit 1).
- 21 becomes 1211 (one 2 followed by one 1).
- 1211 becomes 111221 (one 1, one 2, and two 1s).
- 111221 becomes 312211 (three 1s, two 2s, and one 1).

Write function `lookNsay :: Int → Int` which takes an number and generates the next number in its look and say sequence.

x Triangles (Codeforces)

Pavel has several sticks with lengths equal to powers of two. He has a_0 sticks of length $2^0 = 1$, a_1 sticks of length $2^1 = 2$, ..., a_n sticks of length 2^n .

Pavel wants to make the maximum possible number of triangles using these sticks. The triangles should have strictly positive area, each stick can be used in at most one triangle.

It is forbidden to break sticks, and each triangle should consist of exactly three sticks. Write a function `triangles :: [Int] → Int` to find the maximum possible number of triangles.

Examples

```
triangles [1,2,2,2,2] = 3
triangles [1,1,1] = 0
triangles [3,3,3] = 1
```

In the first example, Pavel can, for example, make this set of triangles (the lengths of the sides of the triangles are listed): $(2^0, 2^4, 2^4)$, $(2^1, 2^3, 2^3)$, $(2^1, 2^2, 2^2)$.

In the second example, Pavel cannot make a single triangle.

In the third example, Pavel can, for example, create this set of triangles (the lengths of the sides of the triangles are listed): $(2^0, 2^0, 2^0)$, $(2^1, 2^1, 2^1)$, $(2^2, 2^2, 2^2)$.

x Thanos Sort (Codeforces)

Thanos sort is a supervillain sorting algorithm, which works as follows: if the array is not sorted, snap your fingers* to remove the first or the second half of the items, and repeat the process.

Given an input list, what is the size of the longest sorted list you can obtain from it using Thanos sort? Write function `thanos :: Ord a ⇒ [a] → Int` to determine that.

* Infinity Gauntlet required.

Examples

```
thanos [1,2,2,4] = 4
thanos [11, 12, 1, 2, 13, 14, 3, 4] = 2
thanos [7,6,5,4] = 1
```

In the first example the list is already sorted, so no finger snaps are required.

In the second example the list actually has a subarray of 4 sorted elements, but you can not remove elements from different sides of the list in one finger snap. Each time you have to remove either the whole first half or the whole second half, so you'll have to snap your fingers twice to get to a 2-element sorted list.

In the third example the list is sorted in decreasing order, so you can only save one element from the ultimate destruction.

x Deadfish

Deadfish XKCD is a fun, unusual programming language. It only has one variable, called *s*, which starts at 0. You change *s* by using simple commands.

Your task is to write a Haskell program that reads Deadfish XKCD code from a file, runs it, and prints the output.

Deadfish XKCD has the following commands:

Command	What It Does
x	Add 1 to <i>s</i> .
k	print <i>s</i> as a number.
c	Square <i>s</i> ,
d	Subtract 1 from <i>s</i> .
X	Start defining a function (the next character is the function name).
K	Print <i>s</i> as an ASCII character.
C	End the function definition or run a function.
D	Reset <i>s</i> back to 0.
{}	Everything inside curly braces is considered a comment.

Extra Rules:

- *s* must stay between 0 and 255.
- If *s* goes above 255 or below 0, reset it back to 0.
- Ignore spaces, newlines, and tabs in the code.
- Other characters (not commands) work differently depending on the subtask.

While you can do the whole exercise in one go, we recommend doing the following subtasks in order.

1. Basic (x, k, c, d only)

```
run "xkcxdk" = "54"
```

2. Extended (add K, D)

```
run "xkcxxxxcxxxxxxxK" = "H"
```

3. Functions (all commands)

```
run "XUxkCxxCUCUCU" = "345"
```

4. Comments (ignore {})

Ignore content inside curly braces.

Hint: Don't be afraid to use tuples!

x Weakness and Poorness (Codeforces)

You are given a sequence of n integers a_1, a_2, \dots, a_n .

Write a function `solve :: [Int] → Float` to determine a real number x such that the weakness of the sequence $a_1 - x, a_2 - x, \dots, a_n - x$ is as small as possible.

The weakness of a sequence is defined as the maximum value of the poorness over all segments (contiguous subsequences) of a sequence.

The poorness of a segment is defined as the absolute value of sum of the elements of segment.

Examples

```
solve [1,2,3] = 2.0
solve [1,2,3.4] = 2.5
```

Note For the first case, the optimal value of x is 2 so the sequence becomes $-1, 0, 1$ and the max poorness occurs at the segment -1 or segment 1 .

For the second sample the optimal value of x is 2.5 so the sequence becomes $-1.5, -0.5, 0.5, 1.5$ and the max poorness occurs on segment $-1.5, -0.5$ or $0.5, 1.5$.

Polymorphism and Higher Order Functions

§ 6.1. Polymorphism

§ 6.1.1. Classification has always been about *shape* and *behaviour* anyway

Functions are our way, to interact with the elements of a type, and one can define functions in one of the two following ways:

1. Define an output for every single element.
2. Consider the general property of elements, that is, how they look like, and the functions defined on them.

And we have seen how to define functions from a given type to another given type using the above ideas, for example:

`nand` is a function that accepts 2 `Bool` values, and checks if it at least one of them is `False`. We will show two ways to write this function.

The first is too look at the possible inputs and define the outputs directly:

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

The other way is to define the function in terms of other functions and how the elements of the type `Bool` behave

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

The situation is something similar, for a lot of other types, like `Int`, `Char` and so on.

But with the addition of the `List` type from the previous chapter, we were able to add *shape* to the elements of a type, in the following sense:

Consider the type `[Integer]`, the elements of these types are lists of integers, the way one would interact with these would be to treat it as a collection of objects, in which each element is an integer.

- A function for lists would thus have 2 components, at least conceptually if not explicit in the code itself:
 - The first being that of a list, which can be interacted with using functions like `head`.
 - The second being that of `Integer`, So that functions on `Integer` can be applied to the elements of the list.

consider the following example:

```
λ squaring all elements of a list
squareAll :: [Integer] → [Integer]
squareAll [] = []
squareAll (x : xs) = x * x : squareAll xs
```

Here, in the definition when we match patterns, we figure out the shape of the list element, and if we can extract an integer from it, then we square it and put it back in the list.

Something similar can be done with the type `[Bool]`:

- Once again, to write a function, one needs to first look at the *shape* an element as a list, Then pick elements out of them and treat them as `Bool` elements.
- An example of this will be the `and` function, that takes in a collection of `Bool` and returns `True` if and only if all of them are `True`.

```
λ and
and :: [Bool] → Bool
and [] = True -- We call scenarios like this 'vacuously true'
and (x : xs) = x && and xs
```

Once again, the pattern matching handles the shape of an element as a list, and the definition handles each item of a list as a `Bool`.

Then we see functions like the following:

- `elem`, which checks in an element belong to a list.
- `(==)`, which checks if 2 elements are equal.
- `drop`, which takes a list and discards a specified about of items in the list from the beginning.

These functions seem to note care about all of the properties (shape and behaviour together) of their inputs.

- The `elem` function wants its inputs to be list does not care about the internal type of list items as long as some notion of equality if defined.
- The `(==)` works on all types where some notion of equality is defined, this is the only behaviour it is interested in. (A counter example would be the type of functions: `Integer → Integer`, and we will discuss why this is the case soon.)
- The `drop` function just cares about the list structre of an element, and does not look at the behaviour of the list items at all.

To define any function in haskell, one needs to give them a type, haskell demands so, so lets look at the case of the `drop` function. One possible way to have it would be to define one for every single type, as shown below:

```
dropIntegers :: Integer → [Integer] → [Integer]
dropIntegers = ...
dropChars :: Integer → [Char] → [Char]
dropChars = ...
dropBools :: Integer → [Bool] → [Bool]
dropBools = ...
.
.
.
```

but that has 2 problems:

- The first is that the definition of all of these functions is the exact same, so doing this would be a lot of manual work, and one would also need to have different name for different types, which is very inconvenient.
- The second, and arguably a more serious issue, is that it stops us from abstracting, abstraction is the process of looking at a scenario and removing information that is not relevant to the problem.
 - An example would be that the `drop` simply lets us treat elements as lists, while we can ignore the type of items in the list.
 - All of Mathematics and Computer Science is done like this, in some sense it is just that.
 - Linear Algebra lets us treat any set where addition and scaling is defined as one *kind* of thing, without worrying about any other structure on the elements.
 - Metric Spaces let us talk about all sets where there is a notion of distance.
 - Differential Equations let us talk about “change” in many different scenarios.

in all of these fields of study, say linear algebra, a theorem generally involves working with an object, whose exact details we don't assume, just that it satisfies the conditions required for it to be a vector space and seeing what can be done with just that much information.

- And this is a powerful tool because solving a problem in the *abstract* version solves the problem in all *concretized* scenarios.

📖 John Locke, *An Essay Concerning Human Understanding* (1690)

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:

1. Combining several simple ideas into one compound one, and thus all complex ideas are made.
2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.
3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

One of the ways abstraction is handled in Haskell, and a lot of other programming languages is Polymorphism.

≡ Polymorphism

A **polymorphic** function is one whose output type depends on the input type. Such a property of a function is called **polymorphism**, and the word itself is ancient greek for *many forms*.

A polymorphic function differs from functions we have seen in the following ways:

- It can take input from multiple different input types (not necessarily all types, restrictions are allowed).
- Its output type can be different for different inputs types.

An example for such a function that we have seen in the previous section would be:

```
λ drop
drop :: Integer → [a] → [a]
drop _ [] = []
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs
```

The polymorphism of this function is shown in the type `drop :: Integer → [a] → [a]` where we have used the variable `a` (usually called a type variable) instead of explicitly mentioning a type.

The goal of polymorphic functions is to let us abstract over a collection of types. That take a collection of types, based on some common property (either shape, or behaviour, maybe both) and treat that as a collection of elements. This lets us build functions that work on “all lists” or “all maybe types” and so on.

The example `λ drop` brings together all types of lists and only looks at the *shape* of the element, that of a list, and does not look at the behaviour at all. This is shown by using the type variable `a` in the definition, indicating that we don't care about the properties of the list items.

✕ Datatypes of some list functions

A nice exercise would be to write the types of the following functions defined in the previous section: `head`, `tail`, `(!!)`, `take` and `splitAt`.

We have now given a type to one of the 3 functions discussed above, by giving a way to group together types by their common *shape*. This is not enough to give types of the other two functions (`(==)` and `elem`), to do so we define the following:

÷ Behaviour

Given a type `T`, the *behaviour* of the elements in `T` is the set of definable functions whose type includes `T`.

We use this to define the two types of polymorphism, one of which we have already seen in this section, and we will look at the other one more deeply in the next.

÷ 2 Types of Polymorphism

- Polymorphism done by grouping types that with common *shape* is called *Parametric Polymorphism*.
- Polymorphism done by grouping types that with common *behaviour* is called *Ad-Hoc Polymorphism*.

We will come back to parametric polymorphism in the second half of the chapter, but for now we discuss Ad-Hoc polymorphism.

§ 6.1.2. A Taste of Type Classes

Consider the case of the `Integer` functions

```
f :: Integer → Integer
f x = x^2 + 2*x + 1

g :: Integer → Integer
g x = (x + 1)^2
```


We know that both functions, do the same thing in the mathematical sense, given any input, both of them have the same output, so mathematicians call them the same, and write $f = g$ this is called function extensionality. But does the following expression make sense in haskell?

λ Function Extensionality

```
f == g
```

This definitely seems like a fair thing to ask, as we already have a definition for equality of mathematical functions, but we run into 2 issues:

- Is it really fair to say that? In computer science, we care about the way things are computed, that is where the subject gets its name from. A lot of times, one will be able to distinguish between functions, by simply looking at which one works faster or slower on big inputs, and that might be something people would want to factor into what they mean by “sameness”. So maybe the assumption that 2 functions being equal pointwise imply the functions are equal is not wise.
- The second is that in general it is not possible, in this case we have a mathematical identity that lets us prove so, but given any 2 function, it might be that the only way to prove that they are equal would be to actually check on every single value, and since domains of functions can be infinite, this would simply not be possible to compute.

So we can't have the type of `(==)` to be `a → a → Bool`. In fact, if I try to write it, the haskell compiler will complain to me by saying

```
funext.hs:8:7: error: [GHC-39999]
    • No instance for 'Eq (Integer → Integer)' arising from a use of '=='
      ... more error
8 | h = f == g
```

To tackle the problem of giving a type for `(==)`, we define the following:

÷ Typeclasses

Typeclasses are a collection of types, characterized by the common *behaviour*.

The previous section talked about grouping types together by the common *shape* of the elements but **λ Function Extensionality** tells us that there are other properties shared by elements of different types, which we call their *behaviour*. By that we mean the functions that are defined for them.

Typeclasses are how one expresses in haskell, what a collection of types looks like, and the way to do so is by defining the common functions that work for all of them. Some examples are:

- `Eq`, which is the collection of all types for which the function `(==)` is defined.
- `Ord`, which is the collection of all types for which the function `(<)` is defined.
- `Show`, which is the collection of all types for which there is a function that converts them to `String` using the function `show`.

Note that in the above cases, defining one function lets you define some other functions, like `(/=)` for `Eq` and `(<=)`, `(>=)` and others for the `Ord` typeclass.

Now we come back to the `elem` function, the goal of this function is to check if a given element belongs to a list. And the following is a way to write it:

```
elem _ [] = False
elem e (x : xs) = e == x || elem e xs
```

Now lets try to give this a type.

First we see that the `e` must have the same types as the items in the list, but if we try to give it the type

```
elem :: a -> [a] -> Bool
```

we will encounter the same issue as we did in [λ Function Extensionality](#), because of `(==)`. We need to find a way to say that `a` belongs to the collection `Eq`, and this leads to the correct type:

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

x Checking if a list is sorted

Write the function `isSorted` which takes in a list as an argument, such that the elements of the list have a notion of ordering between them, and the output should be true if the list is in an ascending order (equal elements are allowed to be next to each other), and false otherwise.

x Shape is behaviour?

The two types of polymorphism, that is parametric and ad-hoc, are not exclusive, there are plenty of function where both are seen together, an example would be `elem`.

These two happen to not be that different conceptually either, we give elements their *shape* using functions, try figuring out what the functions are for list types, maybe type, tuples and either type.

That being said, the syntax used to define parametric polymorphism sets us to set operations while defining the type of the function which is very powerful.

§ 6.2. Higher Order Functions

One of the most powerful features of functional programming languages is that it lets one pass in functions as argument to another function, and have functions return other functions as outputs, these kinds of functions are known as:

≡ Higher Order Functions

A **higher order function** is a function that does at least one of the following things:

- It takes one or more functions as its arguments.
- It returns a function as an argument.

This is again a way of generalization and is very handy, as we will see in the rest of the chapter.

§ 6.2.1. Currying

Perhaps the first place where we have encountered higher order functions is when we defined `(+)` :: `Int -> Int -> Int` way back in §3.4.. We have been suggesting to think of the type as

`(+) :: (Int, Int) -> Int`, because that is really what we want the function to do, but in haskell

it would actually mean `(+) :: Int → (Int → Int)`, which says the function has 1 interger argument, and it returns a function of type `Int → Int`.

An example from mathematics would be finding the derivative of a differentiable function f at a point x . This is generally represented as $f'(x)$ and the process of computing the derivative can be given to have the type

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

Here $(\mathbb{R} \rightarrow \mathbb{R})^d$ is the type of real differentiable functions.

But one can also think of the derivative operator, that takes a differentiable function f and produces the function f' , which can be given the following type:

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

In general, we have the following theorem:

Theorem Currying: Given any sets A, B, C , there is a *bijection* called *curry* between the sets $C^{A \times B}$ and the set $(C^B)^A$ such that given any function $f : C^{A \times B}$ we have

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

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

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

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

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

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

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

x Exercise

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

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

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

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

And the above theorem is also “implemented” in *haskell* using the following functions:

λ curry and uncurry

```
curry :: ((a, b) → c) → a → b → c
curry f a b = f (a, b)

uncurry :: (a → b → c) → (a, b) → c
uncurry g (a, b) = g a b
```

Currying lets us take a function with with argument, and lets us apply the function to each of them one at a time, rather than applying it on the entire tuple at once. One very interesting result of that is called partial application.

Partial applicaion is precisely the process of fixing some arugments to get a function over the remain- ing, let us look at some examples

```
suc :: Integer → Integer
suc = (+ 1) -- suc 5 = 6

-- | curry examples
neg :: Integer → Integer
neg = (-1 *) -- neg 5 = -5
```

We will find many more examples in the next section.

§6.2.2. Functions on Functions

We have already seen examples of a couple of functions whose arguments themselves are functions. The most recent ones being **λ curry and uncurry**, both of them take functions as inputs and return functions as outputs (note that our definition takes in functions and values, but we can always use partial application), these functions can be thought of as useful operations on functions.

Another very useful example, that a lot of us have seen is composition of functions, when we allow functions as inputs, composition can be treated like a function:

λ composition

```
(.) :: (b → c) → (a → b) → (a → c)
g . f = \a → g (f a)

-- example
square :: Integer → Integer
square x = x * x

-- checks if a number is the same if written in reverse
is_palindrome :: Integer → Bool
is_palindrome x = (s == reverse s)
  where
    s = show x -- convert x to string

is_square_palindrome :: Integer → Bool
is_square_palindrome = is_palindrome . square
```

Breaking a complicated function into simpler parts, and being able to combine them is fairly standard problem solving strategy, in both Mathematics and Computer Science, and in fact in a lot more general scenarios too! Having a clean notation for a tool that used fairly frequently is always a good idea!

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

would have been significantly less impressive if say it was only composition between functions from `Integer → Integer` and `Integer → Bool`.

Another similar function that makes writing code in haskell much cleaner is the following:

```
λ function application function
($ :: (a → b) → a → b
f $ a = f a

(& :: a → (a → b) → b
a & f = f a
```

These may seem like a fairly trivial function that really doesn't offer anything apart from an extra `$`, but the following 3 lines make them useful

```
λ operator precedence
-- The 'r' in infixr says a.b.c.d is interpreted by haskell as a.(b.(c.d))
infixr 9 .
infixr 0 $
infixl 1 &
```

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

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

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

-- also
x & f & g & h & i
```

which in my opinion is much simpler to read!

x Exercise

Write a function `apply_n_times` that takes a function `f` and an argument `a` along with a natural number `n` and applies the function `n` times on `a`, for example: `apply_n_times (+1) 5 3` would return `8`. Also figure out the type of the function.

§ 6.2.3. A Short Note on Type Inference

Haskell is a statically typed language. What that means is that it requires the types for the data that is being processed by the program, and it needs to do so for an analysis that happens before running called type checking.

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

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

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

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

§6.2.4. Higher Order Functions on Maybe Type : A Case Study

The Maybe Type, as defined in Chapter 3 is another playground for higher order functions.

As a refresher on Maybe Types, given a type `a`, one can add an *extra element* to it by making it the type `Maybe a`. For example, given the type `Integer`, whose elements are all the integers, the type `Maybe Integer` will be the collection of integers along with an extra element, which we call `Nothing`.

Maybe Types are meant to capture failure, for example, the `λ function to a maybe type` defines the `reciprocal` function, which takes a rational number, and returns its reciprocal, except when the input is `0`, in which case it returns the *extra value* which is `Nothing`.

To state that elements belong to a Maybe Type they are decorated with `Just`. For example:

- The type of `5` is `Integer`
- The type of `Just 5` is `Maybe Integer`.

To see an example of some functions that use `Maybe` in their type definitions are:

- A safe version of `head` and `tail`:
 - `safeHead :: [a] → Maybe a`
 - `safeTail :: [a] → Maybe [a]`
- A safe way to index a list, that is a safe version of `(!!)`:
 - `safeIndex :: [a] → Int → Maybe a`

Safety First

Define the functions `safeHead`, `safeTail` and `safeIndex`.

Something that should be noted is that so far in the book, `head`, `tail` and `(!!)` are the only functions for which we need safe versions. This is because these are the only functions that are not defined for all possible inputs and can hence give an error while the program executes (that would be like passing empty list to `head`, or indexing an element at a negative position). Every other function we have seen will always have a valid output, that is, it is literally impossible for functions to fail for not having a valid input if one only uses safe functions!

This may seem like a fairly trivial fact for those who are learning haskell as thier first programming language, but for those who has programmed in languages like Java, Python, C or so on, it is impossible to write a program that would lead to an error which is equivalent to the following:

- Nonetype does not have this attribute: Python
- Null Pointer Exception: Java
- Memory Access Violation or Segfault for derefencing a null pointer: C

If these errors have haunted you, you have our condolences, all of these would have been completely avoided if the language had some version of `Maybe`, or even some bare bones type system in case of python.

All of the safety provided by `Maybe` types has 1 potential drawback: When using `Maybe` types, one eventually runs into a problem that looks something like this:

- While solving a complicated problem, one would break it down into simpler parts, that would correspond to many tiny functions, that will come together to form the functions which solve the problem.
- Turns out that one of the functions, maybe something in the very beginning returns a `Maybe Integer` instead of an `Integer`.
- This means that the next function along the chain, would have had to have its input type as `Maybe Integer` to account for the potentially case of `Nothing`.
- This also forces the output type to be a `Maybe` type, this makes sense, if the process fails in the beginning, one might not want to continue.
- The `Maybe` now propagates in this manner through a large section of your code, this means that a huge chunk of code needs to be rewritten to look something like:

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

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

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

This is still not a very elegant way to write things though, and it's just a lot of repetitive work (all of it is just book keeping really, one isn't really adding much to the program by making these changes, except for safety, programmers usually like to call it boilerplate.)

Instead of going and modifying each function manually, we make a function modifier, which is precisely what a higher order function: Our goal, which is obvious from the problem:

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

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

(<$>) :: (a -> b) -> Maybe a -> Maybe b -- symbol version
f <$> a = maybeMap f a

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

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

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

So consider the following chain of functions:

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

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

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

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

x Beyond map

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

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

```
f :: a → Maybe b
g :: b → Maybe c

h x = g <$> f x :: a → Maybe (Maybe c)
```

This is most likely undesirable, the point of `Maybe` was to say that there is a possibility of error, the point of `(<$>)` was to propagate that possible error then the type `Maybe (Maybe c)` seems to not have a place here.

To rectify this, we find a way to compose such functions together:

```
maybe_comp :: (a → Maybe b) → (b → Maybe c) → (a → Maybe c)
infixr 8 ==>
```

This cute looking function is called the fish operator. This will be our way to compose functions of the shape `a → Maybe b` together, but note that the order of inputs is reversed, so it not looks like a pipe through which the value is passed. The above function `h` is defined as follows:

```
h = f ==> g :: a → Maybe c
```

This function, takes a value of type `a`, first applies `f` to it, and then applies `g` to it in a way that the final output is of type `Maybe c`, and of course, we can use this to make longer chains!

```
func1 :: a → Maybe b
func2 :: b → Maybe c
func3 :: c → Maybe d
func4 :: d → Maybe e

final :: a → Maybe e
final = func1 ==> func2
      ==> func3 ==> func4
```

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

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

Advanced List Operations

§7.1. List Comprehensions

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

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

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

which can be written in Haskell as

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

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

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

⚠ Defining map using pattern matching and list comprehension

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

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

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

⚠ Defining filter using pattern matching and list comprehension

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

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

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

⚡ Defining cartesian product using pattern matching and list comprehension

```

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

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

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

```

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

⚡ A naive way to get pythagorean triplets

```

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

```

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

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

⚡ A mid way to get pythagorean triplets

```

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

```

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

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

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

⚡ The optimal way to get pythagorean triplets

```

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

```

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

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

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

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

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

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

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

So we can implement `merge`, rather simply as

λ The merge function of mergesort

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

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

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

That means, we can now merge sort using the function

ⓧ An implementation of mergesort

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

ⓧ MergeSort Works?

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

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

(ii) Prove that `mergeSort` works using induction on the size of list to be sorted.

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

This implies

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

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

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

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

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

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

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

ⓧ An implementation of Quick Sort

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

ⓧ Quick Sort works?

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

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

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

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

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

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

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

ⓧ A Strange Proof

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

x Faster Quick Sort

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

Try to figure out this implementation.

§7.2. Zip it up!

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

x The boring zip

Haskell has an inbuilt function called `zip`. Its behaviour is as follows

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

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

The solution to the above exercise is, rather simply:

Implementation of zip function

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

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

Implementation of zipWith function

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

x Alternate definitions

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

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

The zipWith fibonacci

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

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

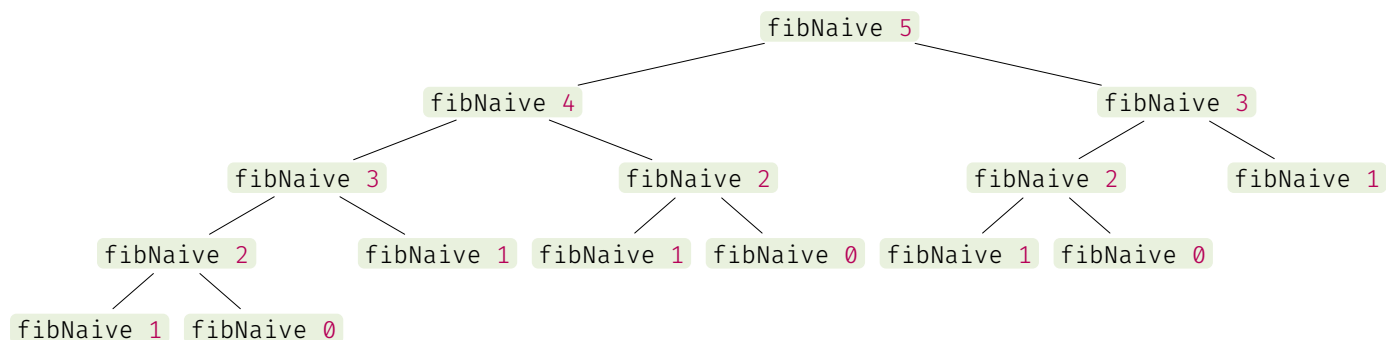
1. Since by definition `fibs = 0 : 1 : (something)`, the first element is `0`.

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

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

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

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



And one can easily see that we make a bunch of unnecessary recomputations, and thus a lot of unnecessary additions. On the other hand, using our `zipWith` method, only computes things once, and hence makes only as many additions as required.

x Exercise

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

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

x Tromino's Pizza I

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

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

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

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

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

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

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

Tromino's Pizza II

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

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

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

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

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

Exercise

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

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

x Tromino's Pizza III

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

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

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

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

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

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

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

x Composing zipWith's

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

x Unzip

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

Try to figure out the implementation of `unzip`.

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

§7.3.1. Orgami of Code!

A lot of recursion on lists has the following structure

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

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

Some common examples from the inbuilt functions are:

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

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

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

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

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

Definition of foldr

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

This shortens our definitions to

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

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

Definition of foldr1

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

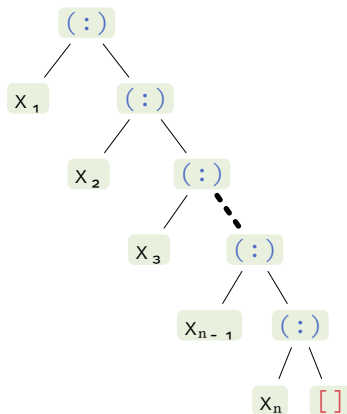
Like we could now define `product = foldr1 (*)` which is much more clean as we don't have to define a weird vacuous case.

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

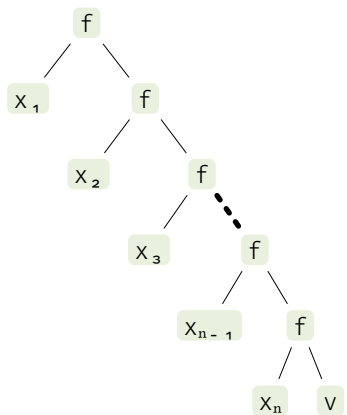
```
foldr v f [1,2,3,4] = 1 `f` (2 `f` (3 `f` (4 `f` v)))
-- and if f is right associative
= 1 `f` 2 `f` 3 `f` 4 `f` v
```

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

Using trees, A list can be represented in the form



which is converted to:



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

The recursive pattern

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

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

Definition of foldl and foldl1

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

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

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

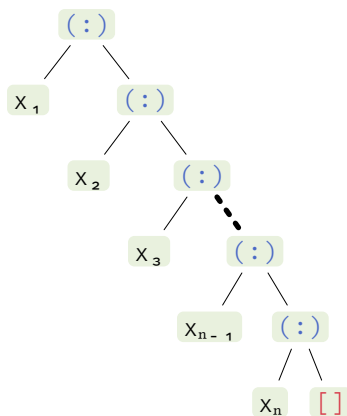
```
sum      = foldl (+) 0
product  = foldl (*) 1
or       = foldl (||) False
and      = foldl (&&) True
```

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

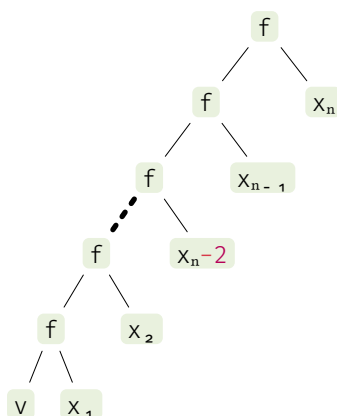
The computation of `foldl` proceeds like

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

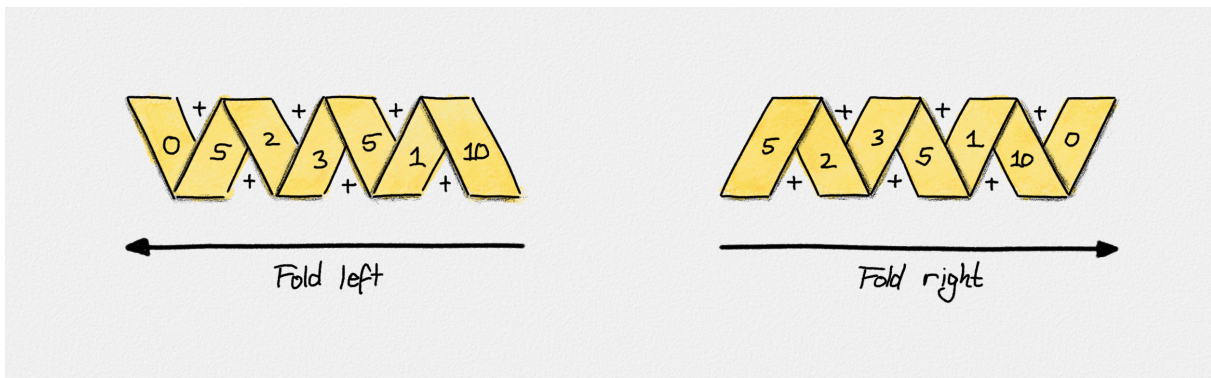
Or in tree form as



which is converted to:



Another very cute picture to summarize the differences is:



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

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

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

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

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

Instead, can we use something we studied in ch-6? Maybe.⁹

Implementation of `unfoldr`

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

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

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

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

While `foldr` and `foldl` are some of the most common favorite function of haskell programmers; `unfoldr` remains mostly ignored. It is so ignored that to get the inbuilt version, one has to `import Data.List`.

⁹Pun intended.

We will soon see an egregious case where Haskell's own website ignored it. One of the paper we referred was literally titled "The Under-Appreciated Unfold".

x Some more inbuilt functions

Implement the following functions using fold and unfold.

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

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

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

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

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

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

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

x Base Conversion

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

(ii) Given a number in base 10, convert to a list of digits in base `k`.

`num2lis :: Int → Int → [Int]` with the usage `num2list base numberInBase10`

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

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

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

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

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

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


X A list of Primes

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

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

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

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

A list of primes using `unfoldr`

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

X Subsequences

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

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

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

Please fill in the blanks below

A naive, non-infinite compatible definition is:

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

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

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

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

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

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

Substituting all thins through the definition.

⚠ Space to write the definition of sublists

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

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

ⓧ The Recap Problem (Euler's Project 268)

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

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

Hint : Think about PIE but not π .

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

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

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

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

Another type of function we sometimes want to define are:

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

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

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

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

x Ackerman Function

The Ackerman function is defined as follows:

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

Define this in one single line using `foldr`.

Let's say `foldr f v ≅ ack`.

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

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

Let's say `foldr g w ≅ ack (x:xs)`.

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

This means `w = ack xs [1]` and

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

Thus, `g = (\y z → ack xs z)`.

And finally, now working towards `f`, we get

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

This gives us the definition

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

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

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

x Removing duplicates

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

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

x More dropping and more taking

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

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

Implement them using recursion and then using folds.

§7.3.2. Numerical Integration

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

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

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

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

Naive Integration

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

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

Naive Integration without repeated computation

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

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

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

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

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

This means we can improve our sequence by eliminating the error

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

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

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

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

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

This allows us to improve our sequence

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

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

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

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

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

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

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

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

⚡ An optimized function for numerical integration

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

With this we are done!

⚡ Simpson's Rule

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

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

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

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

§7.3.3. Time to Scan

We will now talk about folds lesser known cousing scans.

÷ Scans

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

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

and very much similarly as

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

There are also very much similar `scanr1` and `scanl1`.¹⁰

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

¹⁰Similar to our note in `fold`, there is a function pair `scanl'` and `scanr1'`, which similar to `foldl'` and `foldr1'`, and have the same set of benefits. This makes them the defaults, but similarly, to understand them well, we need to discuss how Haskell's lazy computation actually works and the way to bypass it. This is done in chapter 9.

```

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

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

```

x Scan as a fold

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

x Defining `scanl1` and `scanr1`

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

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

x Not Quite Lisp (AOC 2015, 1)

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

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

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

For example:

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

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

This is quite simple using folds.

```

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

```

But every AOC question always has a part 2!

x Not Quite Lisp, 2

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

For example:

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

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

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

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

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

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

x Perfectly Spherical Houses in a Vacuum (AOC 2015)

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

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

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

For example:

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

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

x Perfectly Spherical Houses in a Vacuum II

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

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

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

For example:

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

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

We will also briefly talk about something called Segmented Scan.

÷ Segmented Scan

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

For example, a segmented looks like is:

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

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

The implementation of function is as follows

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

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

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

§7.4. Exercises

x Factors

(i) Write an optimized function `factors :: Int → [Int]` which takes in an integer and provides a list of all its factors.

(ii) Write an optimized function `primeFactors :: Int → [Int]` which takes in an integer and provides a list of all its prime factors, repeated wrt to multiplicity. That is `primeFactors 100 = [2,2,5,5]`.

x Trojke (COCI 2006, P3)

Mirko and Slavko are playing a new game, Trojke (Triplets). First they use a chalk to draw an $N \times N$ square grid on the road. Then they write letters into some of the squares. No letter is written more than once in the grid.

The game consists of trying to find three letters on a line as fast as possible. Three letters are considered to be on the same line if there is a line going through the centre of each of the three squares (horizontal, vertical and diagonal).

After a while, it gets harder to find new triplets. Mirko and Slavko need a program that counts all the triplets, so that they know if the game is over or they need to search further.

Write a function `trojke :: [String] → Int` which takes the contents of the lines and outputs the number of lines.

Example:

```
trojke [
  "... D",
  ".. C.",
  ".B..",
  "A..."
] = 4

trojke [
  "..T..",
  "A....",
  ".FE.R",
  "...X",
  "S...."
] = 3

trojke [
  "....AB....",
  "..C....D..",
  ".E.....F.",
  "...G..H...",
  "I.....J",
  "K.....L",
  "...M..N...",
  ".O.....P.",
  "..Q....R..",
  "....ST...."
] = 0
```

x Deathstar (COCI 2015, P3)

Young jedi Ivan has infiltrated in The Death Star and his task is to destroy it. In order to destroy The Death Star, he needs an array A of non-negative integers of length N that represents the code for initiating the self-destruction of The Death Star. Ivan doesn't have the array, but he has a piece of paper with requirements for that array, given to him by his good old friend Darth Vader.

On the paper, a square matrix of the size is written down. In that matrix, in the row i and column j there is a number that is equal to bitwise and between numbers a_i and a_j . Unfortunately, a lightsaber has destroyed all the fields on the matrix's main diagonal and Ivan cannot read what is on these fields. Help Ivan to reconstruct an array for the self-destruction of The Death Star that meets the requirements of the matrix.

The solution doesn't need to be unique, but will always exist. Your function `destroy :: [[Int]] → [Int]` only needs to find one of these sequences.

Example:

```
destroy [
  [0,1,1],
  [1,0,1],
  [1,1,0]] = [1,1,1]
destroy [
  [0,0,1,1,1]
  [0,0,2,0,2]
  [1,2,0,1,3]
  [1,0,1,0,1]
  [1,2,3,1,0]] = [1,2,3,1,11]
```

Extra Credit : There is a way to do this in one line.

X Nucleria (CEOI 2015 P5)

Long ago, the people of Nucleria decided to build several nuclear plants. They prospered for many years, but then a terrible misfortune befell them. The land was hit by an extremely strong earthquake, which caused all the nuclear plants to explode, and radiation began to spread throughout the country. When the people had made necessary steps so that no more radiation would emanate, the Ministry of Environment started to find out how much individual regions were polluted by the radiation. Your task is to write a function `quary :: (Int, Int) → [(Int, Int, Int, Int)] → Float` that will find the average radiation in Nucleria given data.

Nucleria can be viewed as a rectangle consisting of $W \times H$ cells. Each nuclear plant occupies one cell and is parametrized by two positive integers: a , which is the amount of radiation caused to the cell where the plant was, and b , which describes how rapidly the caused radiation decreases as we go farther from the plant.

More precisely, the amount of radiation caused to cell $C = (x_C, y_C)$ by explosion of a plant in cell $P = (x_P, y_P)$ is $\max(0, a - b \cdot d(P, C))$, where $d(P, C)$ is the distance of the two cells, defined by $d(P, C) = \max(|x_P - x_C|, |y_P - y_C|)$ (i.e., the minimum number of moves a chess king would travel between them).

The total radiation in a cell is simply the sum of the amounts that individual explosions caused to it. As an example, consider a plant with $a = 7$ and $b = 3$. Its explosion causes 7 units of radiation to the cell it occupies, 4 units of radiation to the 8 adjacent cells, and 1 unit of radiation to the 16 cells whose distance is 2.

The Ministry of Environment wants to know the average radiation per cell. The input will be in the form `quary (W, H) [(x1,y1,a1,b1), (x2,y2,a2,b2)]` where we first give the size of Nucleria and then the position of plants and their parameter.

Example:

```
quary (4,3) [(1,1,7,3),(3,2,4,2)] = 3.67
```

The radiation in Nucleria after the two explosions is as follows:

```
7632
4652
1332
```

x Garner's Algorithm

A consequence of the Chinese Remainder Theorem is, that we can represent big numbers using an array of small integers. For example, let p be the product of the first 1000 primes. p has around 3000 digits.

Any number a less than p can be represented as a list a_1, \dots, a_k , where $a_i \equiv a \pmod{p_i}$. But to do this we obviously need to know how to get back the number a from its representation (which we will call the CRT form).

Another form for numbers is called the mixed radix form. We can represent a number a in the mixed radix representation as: $a = x_1 + x_2p_1 + x_3p_1p_2 + \dots + x_kp_1\dots p_{k-1}$ with $x_i \in [0, p_i)$

(i) Make a list of first 1000 primes. Call it `primeThousand :: [Int]`.

(ii) Write function `encode :: Int → [Int]` which encodes a number into the CRT form.

(iii) You had constructed an extremely fast way to compute modulo inverses in x [Modulo Inverse](#). Use it to create `residue :: [[Int]]` such that r_{ij} denotes the inverse of p_i modulo p_j .

(iv) Garner's algorithm converts from the CRT form to the mixed radix form. We want you to implement `garner :: [Int] → [Int]`. The idea of the algorithm is as follows:

Substituting a from the mixed radix representation into the first congruence equation we obtain

$$a_1 \equiv x_1 \pmod{p_1}.$$

Substituting into the second equation yields

$$a_2 \equiv x_1 + x_2p_1 \pmod{p_2},$$

which can be rewritten by subtracting x_1 and dividing by p_1 to get

$$\begin{aligned} a_2 - x_1 &\equiv x_2p_1 \pmod{p_2} \\ (a_2 - x_1)r_{12} &\equiv x_2 \pmod{p_2} \\ x_2 &\equiv (a_2 - x_1)r_{12} \pmod{p_2} \end{aligned}$$

Similarly we get that

$$x_3 \equiv ((a_3 - x_1)r_{13} - x_2)r_{23} \pmod{p_3}.$$

(v) Finally, now write a function `decodeMixed :: [Int] → Int` which decodes from the mixed radix form.

(vi) Finally, combine these functions and write a `decode :: [Int] → Int` which decodes from CRT form.

Note : We find it extremely cool to know that so much of math goes on in simply representing big integers in high accuracy systems. Like from airplane cockpits to rocker simulations to some games like Valorent, a very cool algorithm is keeping it up and running.

✕ Shanks Baby Steps-Giant Steps algorithm

A surprisingly hard problem is, given a, b, m computing x such that $a^x \equiv b \pmod m$ efficiently. This is called the discrete logarithm. We will try to walk through implementing an algorithm to do so efficiently. At the end, you are expected to make a function `dlog :: Int → Int → Int → Maybe Int` which takes in a, b and m and returns x such that $a^x \equiv b \pmod m$ if it exists.

Let $x = np - q$, where n is some pre-selected constant (by the end of the description, we want you to think of how to choose n).

We can also see that $p \in \{1, 2, \dots, \lceil \frac{m}{n} \rceil\}$ and $q \in \{0, 1, \dots, n - 1\}$.

p is known as giant step, since increasing it by one increases x by n . Similarly, q is known as baby step. Try to find the bounds

Then, the equation becomes: $a^{np-q} \equiv b \pmod m$.

Using the fact that a and m are relatively prime, we obtain:

$$a^{np} \equiv ba^q \pmod m$$

So we now need to find the p and q which satisfies this. Well, that can be done quite easily, right?

Keeping in mind ✕ [Modular Exponentiation](#), what n should we choose to be most optimal?

X A very cool DP (Codeforces)

Giant chess is quite common in Geraldion. We will not delve into the rules of the game, we'll just say that the game takes place on an $h \times w$ field, and it is painted in two colors, but not like in chess. Almost all cells of the field are white and only some of them are black. Currently Gerald is finishing a game of giant chess against his friend Pollard. Gerald has almost won, and the only thing he needs to win is to bring the pawn from the upper left corner of the board, where it is now standing, to the lower right corner. Gerald is so confident of victory that he became interested, in how many ways can he win?

The pawn, which Gerald has got left can go in two ways: one cell down or one cell to the right. In addition, it can not go to the black cells, otherwise the Gerald still loses. There are no other pawns or pieces left on the field, so that, according to the rules of giant chess Gerald moves his pawn until the game is over, and Pollard is just watching this process.

Write a function `wins :: (Int, Int) → [(Int, Int)] → Int` to compute the number of ways to win on a (w, h) grid with black squares at given coordinates. The pawn starts at $(1, 1)$ and we must go till (w, h) .

Examples

```
wins (3,4) [(2,2),(2,3)] = 2
wins (100,100) [(15,16),(16,15),(99,98)] = 545732279
```

Hint : This is a very hard question to do optimally. The 'standard' way to do so would be making a function `ways :: (Int, Int) → Integer` and counting the ways to every square recursively and setting the black squares as 0.

This is not optimal if we have a huge grid. Here the idea is to sort the black squares lexicographically. Let this sorted list be b_1, b_2, \dots, b_n . We add $b_{n+1} = (w, h)$. Let the paths (ignoring black squares) from $(1, 1)$ to b_i be d_i . Let the paths respecting black squares be c_i . Our goal is to find c_{n+1} .

Also try defining a function `paths :: (Int, Int) → (Int, Int) → Int` which counts the paths from (x_1, y_1) to x_2, y_2 without any black squares. This function should give us d_i 's. Can we use these d_i and the `paths` function to get c_i 's?

x Mars Rover (Codeforces)

If you felt bad that I gave a hint in the last problem, here is a similar problem for you to do all on your own.

Research rover finally reached the surface of Mars and is ready to complete its mission. Unfortunately, due to the mistake in the navigation system design, the rover is located in the wrong place.

The rover will operate on the grid consisting of n rows and m columns. We will define as (r, c) the cell located in the row r and column c . From each cell the rover is able to move to any cell that share a side with the current one.

The rover is currently located at cell $(1, 1)$ and has to move to the cell (n, m) . It will randomly follow some shortest path between these two cells. Each possible way is chosen equiprobably.

The cargo section of the rover contains the battery required to conduct the research. Initially, the battery charge is equal to s units of energy.

Some of the cells contain anomaly. Each time the rover gets to the cell with anomaly, the battery loses half of its charge rounded down. Formally, if the charge was equal to x before the rover gets to the cell with anomaly, the charge will change to $\lfloor \frac{x}{2} \rfloor$.

While the rover picks a random shortest path to proceed, write function `charge :: (Int, Int) → [(Int, Int)] → Int → Float` to compute the expected value of the battery charge after it reaches cell (n, m) , with the anomalies at some positions $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ if we started with some c charge.

Note: If the cells $(1, 1)$ and (n, m) contain anomaly, they also affect the charge of the battery.

Examples

```
charge (3,3) [(2,1),(2,3)] 11 = 6.33333333333
```

x Vegetables (ZCO 2024)

You are a farmer, and you want to grow a wide variety of vegetables so that the people in your town can eat a balanced diet.

In order to remain healthy, a person must eat a diet that contains N essential vegetables, numbered from 1 to N . In total, your town requires A_i units of each vegetable i , for $1 \leq i \leq N$. In order to grow a single unit of vegetable i , you require B_i units of water.

However, you can use upgrades to improve the efficiency of your farm. In a single upgrade, you can do one of the following two actions:

1. You can improve the nutritional value of your produce so that your town requires one less unit of some vegetable i . Specifically, you can choose any one vegetable i such that $A_i \geq 1$, and reduce A_i by 1.
2. You can improve the quality of your soil so that growing one unit of some vegetable i requires one less unit of water. Specifically, you can choose any one vegetable i such that $B_i \geq 1$, and reduce B_i by 1.

If you use at most X upgrades, what is the minimum possible number of units of water you will need to feed your town? Write a function `water :: [Int] → [Int] → Int → Int` to answer this where the first list is A , second is B and the integer is X .

x de Polignac Numbers (Rosetta Code)

Alphonse de Polignac, a French mathematician in the 1800s, conjectured that every positive odd integer could be formed from the sum of a power of 2 and a prime number.

He was subsequently proved incorrect.

The numbers that fail this condition are now known as de Polignac numbers.

Technically 1 is a de Polignac number, as there is no prime and power of 2 that sum to 1. De Polignac was aware but thought that 1 was a special case. However, 127 is also fails that condition, as there is no prime and power of 2 that sum to 127.

As it turns out, de Polignac numbers are not uncommon, in fact, there are an infinite number of them.

- Find and display the first fifty de Polignac numbers.
- Find and display the one thousandth de Polignac number.
- Find and display the ten thousandth de Polignac number.

X

The Bifid cipher is a polygraphic substitution cipher which was invented by Félix Delastelle in around 1901. It uses a 5 x 5 Polybius square combined with transposition and fractionation to encrypt a message. Any 5 x 5 Polybius square can be used but, as it only has 25 cells and there are 26 letters of the (English) alphabet, one cell needs to represent two letters - I and J being a common choice.

Operation Suppose we want to encrypt the message “ATTACKATDAWN”.

We use this archetypal Polybius square where I and J share the same position.

x/y	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

The message is first converted to its x, y coordinates, but they are written vertically beneath.

```
A T T A C K A T D A W N
1 4 4 1 1 2 1 4 1 1 5 3
1 4 4 1 3 5 1 4 4 1 2 3
```

They are then arranged in a row.

```
1 4 4 1 1 2 1 4 1 1 5 3 1 4 4 1 3 5 1 4 4 1 2 3
```

Finally, they are divided up into pairs which are used to look up the encrypted letters in the square.

```
14 41 12 14 11 53 14 41 35 14 41 23
D Q B D A X D Q P D Q H
```

The encrypted message is therefore “DQBDAXDQPDQH”.

Decryption can be achieved by simply reversing these steps.

Write functions in haskell to encrypt and decrypt a message using the Bifid cipher.

Use them to verify (including subsequent decryption):

- The above example.
- The example in the Wikipedia article using the message and Polybius square therein.
- The above example but using the Polybius square in the Wikipedia article to illustrate that it doesn't matter which square you use as long, of course, as the same one is used for both encryption and decryption.

In addition, encrypt and decrypt the message “The invasion will start on the first of January” using any Polybius square you like. Convert the message to upper case and ignore spaces.

x Colorful Numbers (Rosetta Code)

A colorful number is a non-negative base 10 integer where the product of every sub group of consecutive digits is unique.

For example: 24753 is a colorful number. 2, 4, 7, 5, 3, $(2 \times 4)8$, $(4 \times 7)28$, $(7 \times 5)35$, $(5 \times 3)15$, $(2 \times 4 \times 7)56$, $(4 \times 7 \times 5)140$, $(7 \times 5 \times 3)105$, $(2 \times 4 \times 7 \times 5)280$, $(4 \times 7 \times 5 \times 3)420$, $(2 \times 4 \times 7 \times 5 \times 3)840$

Every product is unique.

2346 is not a colorful number. 2, 3, 4, 6, $(2 \times 3)6$, $(3 \times 4)12$, $(4 \times 6)24$, $(2 \times 3 \times 4)48$, $(3 \times 4 \times 6)72$, $(2 \times 3 \times 4 \times 6)144$

The product 6 is repeated.

Single digit numbers are considered to be colorful. A colorful number larger than 9 cannot contain a repeated digit, the digit 0 or the digit 1. As a consequence, there is a firm upper limit for colorful numbers; no colorful number can have more than 8 digits.

- Write a function to test if a number is a colorful number or not.
- Use that function to find all of the colorful numbers less than 100.
- Use that function to find the largest possible colorful number.
- Find and display the count of colorful numbers in each order of magnitude.
- Find and show the total count of all colorful numbers.

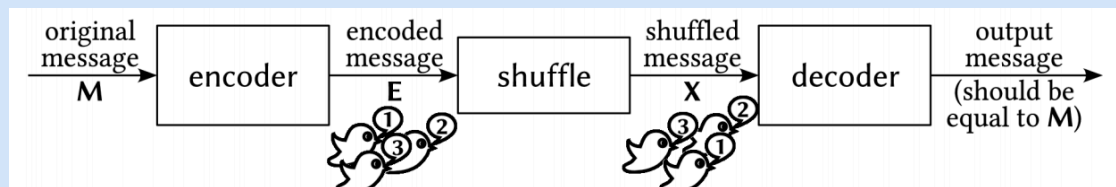
X Data Transfer Protocol on IOI? Parrots (IOI 2011, P6)

Yanee is a bird enthusiast. Since reading about IP over Avian Carriers (IPoAC), she has spent much of her time training a flock of intelligent parrots to carry messages over long distances.

Yanee's dream is to use her birds to send a message M to a land far far away. Her message M is a sequence of N (not necessarily distinct) integers, each between 0 and 255, inclusive. Yanee keeps some K specially-trained parrots. All the parrots look the same; Yanee cannot tell them apart. Each bird can remember a single integer between 0 and R , inclusive.

Early on, she tried a simple scheme: to send a message, Yanee carefully let the birds out of the cage one by one. Before each bird soared into the air, she taught it a number from the message sequence in order. Unfortunately, this scheme did not work. Eventually, all the birds did arrive at the destination, but they did not necessarily arrive in the order in which they left. With this scheme, Yanee could recover all the numbers she sent, but she was unable to put them into the right order.

To realize her dream, Yanee will need a better scheme, and for that she needs your help.



- Try to design a function that receives the Original Message (of some length N with numbers between 0 – 255) and encode the original messages to another message sequence, it's called Encoded Message with limit the size of message must not exceed K and using numbers from 0 – R .
- The encoded message from will be shuffled.
- Receive the Shuffled Message and Decode back to Original Message.

You must implement the `encode :: [Int] → [Int]` and `decode :: [Int] → [Int]` process with you own method. We suggest the following roadmap,

Subtask 1: $N = 8$, all elements of the original message are either 0 or 1, $R = 2^{16} - 1$, $K = 10 * N = 80$.

Subtask 2: $1 \leq N \leq 16$, $R = 2^{16} - 1$, $K = 10 * N$

Subtask 3: $1 \leq N \leq 16$, $R = 255$, $K = 10 * N$

Subtask 4: $1 \leq N \leq 32$, $R = 255$, $K = 10 * N$

Subtask 4: $1 \leq N \leq 32$, $R = 255$, $K = 10 * N$

Subtask 5: We now want you to try to reduce the ratio between encoded message length and original message length upto

$$1 \leq N \leq 64$$

. The mathematical limit for the best ratio one can get is slightly above $\frac{261}{64} \approx 4.08$ (Derive the limit!). Anything below 7 is very good, although we (the authours) are very intrested if someone can find the optimal solution.

Note: When the problem came in IOI, no one found the optimal solution. Furthermore, most solutions which got 100, did so they were optimal on the test cases and not in the general case.

X Broken Device (JOI 2016, Spring Training Camp)

Anna wants to send a 60-bit integer to Bruno. She has a device that can send a sequence of 150 numbers that are either 0 or 1. The twist is that L ($0 \leq L \leq 40$) of the positions of the device are broken and can only send 0. Bruno receives the sequence Anna sent, but he does not know the broken positions.

Anna knows the broken positions but Bruno doesn't. Write functions `encode :: Int → [Int] → [Int]` which given the integer and the broken position encodes the message and function `decode :: [Int] → Int` which decodes the message and recovers the integer sent.

Subtask 1: $K = 0$, This should be very simple as you are just converting to binary.

Subtask 2: $K = 1$, We will have one broken position. If we can somehow indicate the start of our 60 bit sequence, can we find a continuous 61 bit sequence?

Subtask 3: $K = 15$, This is where one needs to be creative. Note $\frac{150}{2} = 75$ and $75 - 15 = 60$. Can you think of something now?

Subtask 4: $K = 40$, The last question had 2, now try with 3 but have some sequences encode more than 1 bit.

X Coins (IOI 2017 Practice)

You have a number C ($0 \leq C < 63$) and an line of 64 coins that are either heads or tails.

As a secret agent, you want to communicate your number to your handler by flipping some of the coins. To avoid being caught, you want to use as few flips as needed. To make the handler aware that you are communicating, you wish to flip at least one coin. (The handler doesn't know the initial sequence of coins).

In the encoding part, you may flip at least one coin and at most K coins of the line.

In the decoding part, you receive the coins already with the changes, in a line, and you must recover the number C .

Write functions `encode :: Int → [Bool] → [Bool]` which takes a number and a list of coins (`True` is heads and `False` is tails) and returns a list of booleans with at least 1 and at most K of them flipped. Write a function `decode :: [Bool] → Int` to recover the number.

Subtask 1: $K = 64$, This is easy.

Subtask 2: $K = 6$, Using our friend binary.

Subtask 3: $K = 1$, Note that bitwise XOR \wedge has some very useful properties. One of these is the fact that it is extremely easy to change and second is that for numbers between $1 - 64$, taking bitwise XOR of some set of numbers will result in a number between $0 - 63$. How can we abuse it?

x Holes (Singapore 2007)

A group of scientists want to monitor a huge forest. They plan to airdrop small sensors to the forest. Due to many unpredictable conditions during airdropping, each sensor will land in a random location in the forest. After all sensors have landed, there will be square regions in the forest that do not contain any sensor. Let us call such a region a hole. It is desirable that all holes are small. This can be achieved by airdropping very large number of sensors. On the other hand, those sensors are expensive. Hence, the scientists want to conduct a computer simulation to determine how many sensors should be airdropped, so that the chances of having a large hole are small.

To conduct this simulation, a function `hole :: [Bool] → Int` is required that, given if a grid of booleans representing the presence of the sensors, outputs the size of the largest hole. This function has to be very efficient since the simulation will be repeated many times with different parameters. You are tasked to write this function.

Example: We will use a matrix of one's and zero's to represent the input for convenience.

$$\text{hole} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = 5$$

as we have a 5×5 grid of 0's, made bold.

x Restorers and Destroyers (Codeforces)

You have to restore a temple in Greece. While the roof is long gone, there are N pillars of marble slabs still standing, the height of the i -th pillar is initially equal to h_i , the height is measured in number of marble slabs. After the restoration all the N pillars should have equal heights.

You are allowed the following operations:

- put a new slab on top of one pillar, the cost of this operation is A ;
- remove a slab from the top of one non-empty pillar, the cost of this operation is R ;
- move a slab from the top of one non-empty pillar to the top of another pillar, the cost of this operation is M .

As the name of the temple is based on the number of pillars, you cannot create additional pillars or ignore some of pre-existing pillars even if their height becomes 0.

What is the minimal total cost of restoration, in other words, what is the minimal total cost to make all the pillars of equal height?

Write a function `cost :: Int → Int → Int → [Int] → Int` which takes the costs A , R and M ; and the list of height of pillars and returns the cost of restoration.

Examples

```
cost 1 100 100 [1,3,8] = 12
cost 100 1 100 [1,3,8] = 9
cost 1 2 4 [5,5,3,6,5] = 4
cost 1 2 2 [5,5,3,6,5] = 3
```

x Numerical Differentiation

Using the techniques described in numerical integration, create a numerical Differentiation function.

You will need to write the following functions. You will be able to reuse some of the definitions from Numerical integration.

```
easyDiff :: (Float → Float) → Float → Float → Float
differentiate :: (Float → Float) → Float → Float → [Float]
elimerror :: Int → [Float] → [Float]
order :: [Float] → Int
improve :: [Float] → [Float]
super :: [Float] → [Float]
within :: Float → [Float] → Float
ans :: (Float → Float) → Float → Float → Float → Float
```


X Cutting Grass

After attempting to program in Grass for the entire morning, you decide to go outside and mow some real grass. The grass can be viewed as a string consisting exclusively of the following characters: `wWv`. `w` denotes tall grass which takes 1 unit of energy to mow. `W` denotes extremely tall grass which takes 2 units of energy to mow. Lastly `v` denotes short grass which does not need to be mowed.

You decide to mow the grass from left to right (beginning to the end of the string). However, every time you encounter a `v` (short grass), you stop to take a break to replenish your energy, before carrying on with the mowing. Your task is to calculate the maximum amount of energy expended while mowing. In other words, write a function `energy :: String → Int` to find the maximum total energy of mowing a patch of grass, that of which does not contain `v`.

Examples

```
energy "WwwvWWWvwwwwwwvWwWw" = 8
energy "w" = 1
energy "W" = 2
energy "vwww" = 3
energy "vWWW" = 6
energy "v" = 0
energy "vvvvvvv" = 0
energy "vwvWvwvWv" = 2
energy "vWWWWWWWWvwwwwwwv" = 21
energy "vWWWWWWWWvwwwwwwv" = 20
energy "vvWvv" = 2
```

X Hacking Cyper (Codeforces)

Polycarpus participates in a competition for hacking into a new secure messenger. He's almost won.

Having carefully studied the interaction protocol, Polycarpus came to the conclusion that the secret key can be obtained if he properly cuts the public key of the application into two parts. The public key is a long integer which may consist of even a million digits!

Polycarpus needs to find such a way to cut the public key into two nonempty parts, that the first (left) part is divisible by a as a separate number, and the second (right) part is divisible by b as a separate number. Both parts should be positive integers that have no leading zeros. Polycarpus knows values a and b .

Write

function `crack :: Integer → Integer → Integer → Maybe (Integer, Integer)` to help Polycarpus and find any suitable method to cut the public key, given the key, a and b ; if possible.

Examples

```
crack 116401024 97 1024 = Just (11640, 1024)
crack 284254589153928171911281811000 1009 1000 = Just (2842545891539,
28171911281811000)
crack 120 12 1 = Nothing
```

x Mohak has explicitly approved this (Codeforces 1874D)

There are $n+1$ cities with numbers from 0 to n , connected by n roads. The i -th road connects city $i-1$ and city i bi-directionally.

After Mohak flew back to city 0 , he found out that he had left her Miku fufu in city n .

Each road has a positive integer level of beauty. Denote the beauty of the i -th road as a_i .

Mohak is trying to find his fufu. As all Miku fans are obsessive indoor creatues and have no sense of direction and are basically useless without Miku fufu for encouragement, he doesn't know which way to go. Every day, he randomly chooses a road connected to the city she currently is in and traverses it.

Let s be the sum of the beauty of the roads connected to the current city. For each road connected to the current city, Mohak will traverse the road with a probability of x/s , where x is the beauty of the road, reaching the city on the other side of the road.

Mohak starts in city 0 and Fufu is in city n .

In order to help Mohak, you want to choose the beauty of the roads such that the expected number of days Mohak takes to find his fufu will be the minimum possible. However, due to limited funding, the sum of beauties of all roads must be less than or equal to m .

Write a function `miku :: Int → Int → Float` which takes the value of m and n and tells us the expected time for Mohak to find his fufu.

Examples

```
miku 3 8 = 5.2
miku 10 98 = 37.721155173329
```

Note In the first example, the optimal assignment of beauty is $a=[1,2,5]$. The expected number of days Mohak needs to get his fufu back is 5.2 .

Hint: Let's say you are given the beauty list a . Now, how will you compute the expected number of days? Consider $f(i)$ to be expected number of days to reach the i -th city. Can you find these recursively? What about $g(i) = f(i) - f(i-1)$? Can you find a form in terms of a , can you use `zipWith` to compute these?

Now, can you make a function to make lists that sum up to m and are of length n ? You can save a lot of time by imposing one condition on the lists. The idea is that we want to keep moving forward.

Introduction to Datatypes

§ 8.1. Datatypes (Once Again)

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

≡ Types 1

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

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

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

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

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

≡ Types 2

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

Some examples of datatypes we have already seen are:

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

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

In §4.1, we saw operations on sets such as

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

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

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

- Types indicate the *shape* of elements and can add information about the functions, for example:
 - `Either [Integer] Bool` tells us that every element of the type is either a list of integers, or a boolean value.
 - `Eq a => a -> [a] -> Maybe Integer` tells us that `a` has a notion of equality defined on it, and the output should be an integer, but the function can potentially fail (that is return `Nothing`).
- Types tell the compiler information about domains and codomains of functions which lets it, to a great extent, check if functions are given inputs they are defined on, and that functions are defined on the values of the domain. (It is not always capable of doing so, for example trying to `sum` an infinite list, but this avoids runtime errors by a lot!)

We will now see how to define our own types.

§8.2. Finite Types

The first step we take in defining types is by creating values and put them together in a collection, this is done using the `data` keyword.

λ **finite types**

```
data Colour = Red | Green | Blue

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

data Coin = Heads | Tails
```

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

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

```
isRed :: Colour -> Bool
isRed Red    = True
isRed Blue   = False
isRed Green  = False
```

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

x Finite Types

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

§8.3. Product Types

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

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

The first way to create a product, which is something we have already seen before is a tuple. As our example we will consider the type `(Integer, Integer)` which we are supposed to interpret as the the set of points on a 2-dimensional lattice. where the two `Integer`s represent the x and y coordinates of a point on the lattice.

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

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

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

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

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

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

```
>>> :t Coord
Coord :: Integer → Integer → Point
```

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

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

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

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

x Dating Profile

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

x Complex Numbers

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

§ 8.4. Parametric Types

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

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

```
Tuple A B = Pair A B

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

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

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

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

§ 8.5. Sum Types

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

- Finite Types

- Viewing constructors as functions
- Parametric Types

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

```
IntOrString = I Integer | S String

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

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

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

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


Better Dating Profile

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

§ 8.6. Inductive Types

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

§ 8.6.1. Inductive Types (as a Mathematician)

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

$$B = \{\text{Set of squares reachable by a bishop (in 0 or more moves)} \\ \text{given that the bottom left square is in the set}\}$$

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

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

One can now create the set B by starting at the bottom left square and one by one adding squares, where the bishop can reach, to set. Here we say that the set B was generated by the bottom left under the bishop movement rules. There was one other important piece of information other than the “base case” and the “operation”, the extra structure imposed by the chess board itself, specifically,

the operations $\langle \text{go top right} \rangle \rightarrow \langle \text{go top left} \rangle$ is the same as $\langle \text{go top left} \rangle \rightarrow \langle \text{go top right} \rangle$ and the board is restricted to an 8×8 grid. This is called generating a set from a value (bottom left square) using a function (bishop movement).

Now consider the following example: You are trying to braid your (or your long haired friend's hair), you do so by starting with 3 bunches of hair (ordered as left, middle and right) and you plan to put them together. This is done by swapping positions of the middle bunch, with either the left or the right bunch (middle one always goes from below), done in an alternating manner. Turns out you're new to this and did not know about the alternating part. And so you start braiding. An important observation you make as you do this every other day is that every different sequence of swaps gives you a different looking braids! (most of them will probably not look good, but you are (or your friend is) a math person, this makes you happy). If we consider B to be the set of all braids that you can create, we can define it using just 2 pieces of information:

- You started with the non-braid which is in the set.
- You either swapped the middle bunch with the left, or with the right bunch and kept doing this.

This is another example of a generated set, but here there are no extra rules other than the starting element and the operations (unlike the restrictions imposed by the geometry of the chessboard). In such cases we say that the set is freely generated (free as in no restrictions). A very useful fact about such sets is that each element can be identified with the sequence of operations used to create them, in fact a lot of the times this is how people talk about elements of freely generated sets.

≡ Freely Generated Sets

Given a collection of **base values** $\mathcal{B} = \{b_1, b_2 \dots b_n\}$ and a collection of **operations** $\mathcal{F} = \{f_1, f_2, \dots f_m\}$ we say that a set S is freely generated from \mathcal{B} using \mathcal{F} if S satisfies the following properties:

- $\mathcal{B} \subseteq S$, that is, all the base values are in the set.
- Given $s_1, s_2 \dots s_n \in S$ and any $f \in \mathcal{F}$ we have that $f(s_1, s_2 \dots s_n) \in S$.
- If there are 2 elements in S constructed as $s_1 = f_1(v_1, v_2 \dots v_p)$ and $s_2 = f_2(w_1, w_2 \dots w_q)$ such that $s_1 = s_2$, then we can say that
 - $f_1 = f_2$,
 - $p = q$ and
 - $v_1 = w_1, v_2 = w_2 \dots v_p = w_q$
- S is the smallest such set satisfying these properties.

Now we see some more examples.

§ 8.6.1.1. Natural Numbers as Inductive Types

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

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

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

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

So we will end up with the set

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

This way is very similar to how mathematicians usually formalize natural numbers.¹¹

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

```
λ nat
data Nat = Z | Succ Nat

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

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

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

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

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

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

Exercise

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

Functions on naturals

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

§8.6.1.2. Lists as Inductive Types

Another interesting example of an inductive type is the set of lists over the type `A`.

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

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

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

¹¹The usually way to define natural numbers was written by Guiseppe Peano and are called the ‘Peano Axioms’ which invole a bunch of rules, whose relevant part can be summarizes as :

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

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

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

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

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

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

Fixing as `A` as `Integer` for now `Nil` represents `[]` and `Cons` takes an integer `n` and gives the constructor `n:`. This is work around to not being able to write infinitely many constructors. The above definition (apart from the syntactic sugar) is how Haskell internally defines lists.

This idea is very much inspired by the concept of Currying which was discussed in §6.2.1..

§8.6.2. (Not Quite) Inductive Types (as a Programmer)

As a programmer, we will be using these types as a *blueprint* for the shape of the element in the type. Specifically to indicate that an element is created by combining other elements of the type together, hence, these are also called recursive datatypes.

In fact, the constructors defined are often used to describe a procedure to check if an element belong to the type, very similar to what we did in \Rightarrow `tree` and in \Rightarrow `well-formed mathematical expression`.

We will see how to extract such a procedure from the constructors of a type in §8.6.2.2..

But first we see a simple example of a recursive datatype, a type to represent arithmetic expressions.

§8.6.2.1. Calculator

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

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

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

λ expr example

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

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

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

λ expr example

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

The following is the procedure to check

x Evaluate and extend

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

Also try extending the Expression type to include more operations.

For those with keen eyes and good memory the shape of `ex` should remind you of the discussion in Why Trees? section in §1.9..

We will now justify the statement made there:

? Ryan Hota, Haskell2025

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

We will now see that Haskell verifies that the element `ex` (from **λ expr example**) is an `Expr` using a procedure that is very similar to what was defined in **≡ tree**.

- We see that the value `ex` is created using the constructor `Add`, so it must produce an `Expr`, now we need to make sure that both of its arguments are also of type `Expr`.
 - The first argument of the above is `Val 3.0` which defines a `Expr`.
 - The second argument of the above constructor is also `Add` so it must produce an `Expr` given that both its arguments are also of type `Expr`.
 - The first argument to this is produced using the constructor `Mul` hence it must produce an element of type `Expr` given the correct arguments.
 - Its first argument is `Val 5.0`

- Its second argument is `Val 10.0`, both of which are `Expr`.
- The second argument to the `Add` is constructed using `Div`, so it must be a tree given both its arguments are `Expr`.
 - Its first argument is constructed using `Exp`, so it must be a `Expr` given its arguments are also `Expr`
 - Its first argument is `Val 8.0`
 - Its second argument is `Val 3.0`, both of which are `Expr`.
 - Its second argument is `Val 2.0` which is a `Expr`.

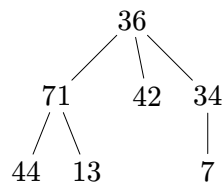
by simply checking that all constructors get inputs of the correct type, Haskell has gone through the procedure defined in `tree` to check that the element `ex` is well defined.

§8.6.2.2. Trees as Inductive Types

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

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

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



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

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

And the above tree can be represented as follows:

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

x Tree Functions

Define the following functions for the tree datatype:

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

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

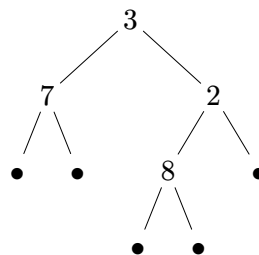
§ 8.6.2.3. Binary Trees

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

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

- Compilers (for functional languages)
- Databases
- Data compression (Huffman Encoding)

A binary tree over integers looks like:



Here unlike the (not necessarily binary) tree, we don't allow leaves to hold values (represented by ●), these allow us to have nodes that behave like they have just 1 child (like 2 in the above example). This also allows the definition to look like the definition of lists:

λ Btree

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

and the example above can be written as:

λ Btree ex

```
bex :: Btree Integer
bex = BNode (BNode 7 Leaf Leaf)
          3
          (BNode (BNode 8 Leaf Leaf)
              2
              Leaf)
```

x Binary Tree Functions

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

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

x All trees are Binary Trees

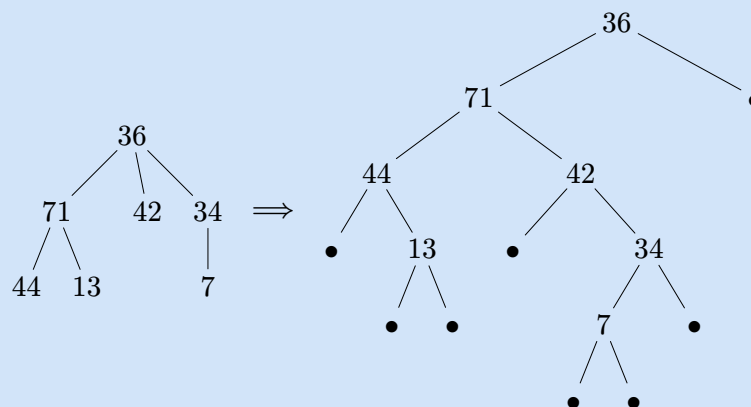
There is a way to convert a (not necessarily binary) tree into a binary tree without losing any information (that is, in a way that one can reconstruct the original tree back from the binary tree).

The idea here is that a binary tree has 2 types of relations between elements, that of a left child, and that of a right child. And we use those to capture the 2 types of relations in trees which are: Being the *first* child, being the *next* sibling. In fact, one can get to each element of a tree from the root by these 2 operations.

So given a tree, we construct its corresponding binary tree as follows:

1. We set the root as also the root of the binary tree.
2. If the node we are looking at in the tree is a leaf and does not have siblings to the right of it, we are done with the node.
3. If the node we are looking at is not a left, let $(x:xs)$ be the list of children. Then we make an edge from the current node to x and we connect x to the next element in the list. That to the one after it and so on.
4. We repeat the previous 2 steps with all new vertices added in step 3 until there are no new nodes left to add.

As an example:



Write a function `treeToBtree :: Tree a → Btree a` to convert to a tree. Also write the function `btreeToTree :: Tree a → Maybe (Btree a)` and show why `Maybe` is required here.

§ 8.6.2.4. Addressing the “Not Quite”...

The title of § 8.6.2. is (Not Quite) Inductive Types (as a Programmer). Turns out that while we have been discussing inductive types this entire section of the chapter, because of the way haskell works, all of the types that we have defined like :

- `Nat`
- `Expr`
- `Tree`
- `Btree`
- `List`

aren't exactly inductive types, a consequence of this is that the type `Nat` isn't exactly the set \mathbb{N} either.

The culprit here is laziness and its exactly what was discussed in §5.11., that is infinite elements, to understand that one of the important points in [Freely Generated Sets](#) is that that the set that is being generated must be the smallest set that satisfies the other properties given in the definition.

Consider the case of the type `Nat = Zero | Succ Nat`, this is supposed to represent the set \mathbb{N} and it satisfies all the properties fo a freely generated set, aka an inductive type except for one: The one about \mathbb{N} being the smallests set that satisfies the given properties because of the following extra elements:

```
infinity :: Nat
infinity = Succ infinity

-- so the element looks like
-- infinity = Succ( Succ( Succ( Succ( Succ( Succ ... )))))
```

So Haskell is letting the `infinity` sneak into our type `Nat`.

The same also holds for all of the other types where we can define

```
inflist :: a → [a]
inflist a = a : inflist a
-- which is simply
-- a : a : a : a : a ...
```

And even infinite expressions like

```
infexpr :: Double → Expr
infexpr x = Add x (infexpr x)
```

which is a funny term that is simply evalutates to

```
infexpr x = x + x + x + x + ...
```

One can make weirder terms that make even less sense than the above and are encouraged to do so, its fun.

There is a formal way to reason about such infinite data structures, and this feature is captured in programming languages like Lean and Agda and is called coinduction, but we will not be discussing it.

We still chose to put emphasis on inductive datatypes as we think that its an idea helpful in designing programs. A lot of the functions that we have discussed so far, like

- `eval` from [Evaluate and extend](#)
- `depth` and `size` from [Tree Functions](#)
- `natToInteger` from [nat and integer](#)

and many more from the list chapter and so on are designed with the idea of inductive types. The purpose of types, as discussed, was to be able to give information to the Haskell compiler about what the functions should expect as an argument, this is one of the places where the Haskell type system fails to express that.

Nonetheless, most of the time people do tend to asssume that the functions are going to get arugments that are finite, which is reasonable in most cases, for example if you are writing a full fledged calculator, using the type described in §8.6.2.1., it would involve something like

- Being able to take a string input

- Parse it into an `Expr` element
- Evaluate it and return the answer

In such cases it is very easy to make sure that expressions are going to be finite (I don't any user has enough free time to enter an infinite expression).

If you are disappointed by the fact that the Haskell compiler is letting a potential error pass through, not that the alternative would be, being able to prove that all programs in Haskell would terminate. This problem is a version of the Halting Problem which is one of the most famous problems in computer science and, in some sense, is the problem that the field of modern computer science stems from. Alan Turing proposed this problem and prove that it is *unsolvable*, so it simply isn't possible for a language like Haskell to check for infinite structures like this. Languages like Lean and Adga achieve this by also disallowing some programs that do terminate. These languages are not Turing Complete (its not possible to write every valid program in this language).

§ 8.7. Same Same but Different

This section is a bit differnt from the previous one, the goal here is not to create new types but to repurpose the old ones.

§ 8.7.1. Same Same

One of the two reasons we had given in § 8.1. for creating types was to make programs make more sense to people who try to read them. Datatypes often indiciate the structure of the code and it doing so, they can express the intent of the programmer. One of the ways in which Haskell makes this easy for us by letting us come up with aliases for types. This is done using the `type` keyword.

📌 `type aliases`

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

type Name = String
type Age = Integer

type Person = (Name, Age)
```

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

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

§ 8.7.2. Different

Another thing one might want to do with an existing type is to use it as your own and define your own functions on it.

One reason you might want to do this would be to use the same datatype to for different pieces of data, and you might not want them to get mixed up. For example, say you're playing a game involving you going down a cave to find treasures. Your 'health' would be an `Integer`, if that goes to `0` your character dies. The 'depth' you're at would also be an `Integer` which would be used to decide how valuable the minerals you find would be. (For those interested, the game in mind is Minecraft).

We can use `data` to let us define separate types like:

```
data Health = Health Integer
data Depth = Depth Integer
```

Now you have 2 copies of `Integer` that haskell will treat differently, making sure that they never get mixed up.

One change that we can make here is to use the `newtype` keyword instead of `data` and get the following:

```
newtype Health = Health Integer
newtype Depth = Depth Integer
```

This just tells Haskell that your datatype has only 1 constructor with exacty 1 field which will allow Haskell to apply some optimizations. It otherwise behaves just like `data` except for the above mentioned restrictions.

Computation as Reduction

Complexity

§10.1. Introduction

< to do >

§10.2. Asymptotics

§10.2.1. Big El

÷ Big Ell notation

The symbol \mathcal{L} means absolutly atmost. For example, $\mathcal{L}(4)$ refers to a value whoose absolute value is less than or equal to 4. For example, $\pi = \mathcal{L}(4)$.

Note, the $=$ sign is not trasitive in this regard. The reason for this is because the notation came from math and unlike Computer scientists, mathematicains have a small vocabulary and couldn't think of another symbol. As a Mathematician once put in words(quite surprising),

De Bruijn

Mathematicians customarily use the $=$ sign as they use the word "is" in English: Aristotle is a man, but a man isn't necessarily Aristotle.

You can also see this by the story, let's say a textbook author's typewriter doesn't have $>$ sign. So they start using $5 = \mathcal{L}(3)$ to repressent that 3 is lesser than 5. They can also write $\mathcal{L}(3) = \mathcal{L}(5)$ but not $\mathcal{L}(5) = \mathcal{L}(3)$

Comming back to mathematics, Big-A notation is very compatible with arithmetic.

Theorem

1. $\pi = 3.14 + \mathcal{L}(0.005)$
2. $10^{\mathcal{L}(2)} = \mathcal{L}(100)$

Proof We will just use the defination of $-k \leq A(k) \leq k$.

Thus, $\pi = 3.14 + \mathcal{L}(0.005) \iff \pi \in [3.14 - 0.05, 3.14 + 0.05]$ which is true as $3.140 < \pi = 3.1415926... < 3.142$.

Similarly, $10^{\mathcal{L}(2)} = \mathcal{L}(100) \iff [10^{-2}, 10^2] \subseteq [-100, 100]$ which is true. ■

Theorem $\mathcal{L}(x) \cdot \mathcal{L}(y) = \mathcal{L}(xy)$

Proof We can eliminate the \mathcal{L} to get that the above statement is equivalent to, $[-xy, xy] \subseteq [-xy, xy]$ which is trivially true. ■

We can use the notation for variables as well,

Theorem

1. $\sin x = \mathcal{L}(1)$;
2. $\mathcal{L}(x) = x \cdot \mathcal{L}(1)$;
3. $\mathcal{L}(x) + \mathcal{L}(y) = \mathcal{L}(x + y)$, where $x, y \geq 0$;
4. $(1 + \mathcal{L}(t))^2 = 1 + 3\mathcal{L}(t)$, where $t = A(1)$.

Proof Left as an exercise to reader. ■

x **Big Ell Analysis**

1. Simplify: $(5 + \mathcal{L}(0.2)) + (3 + \mathcal{L}(0.1))$
2. Simplify: $(5 + \mathcal{L}(0.2)) * (3 + \mathcal{L}(0.1))$
3. Express e^x for $|x| \leq 1$ in the form $1 + \mathcal{L}(u)$ for some $u = kx$. Hint: Use the fact that $e^x = 1 + x + \frac{x^2}{2!} + \dots$
4. Define a sequence $x_0 =, x_n = 2x_{n-1} + A(\varepsilon_n)$ where $\varepsilon_n > -$. Then solve for x_n when
 1. $\varepsilon_n = \frac{1}{3^n}$
 2. $\varepsilon = \frac{1}{n^2}$

§10.2.2. The Hierarchy of Functions

≡ **Asymptotic Dominance**

Asymptotic Dominance is a relation over functions, where $f(n) \prec g(n)$ when $g(n)$ approaches infinity faster than $f(n)$. We can formalize this by saying:

$$f(n) \prec g(n) \iff \limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} = 0$$

Furthermore, $f(n) \asymp g(n) \iff \limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} = k$ where $0 < k < \infty$ is constant.

Theorem Asymptotic dominance is transitive.

Proof Let $f(n) \prec g(n)$ and $g(n) \prec h(n)$. We wish to show $f(n) \prec h(n)$.

$$\limsup_{n \rightarrow \infty} \frac{|f(n)|}{|h(n)|} = \limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} \cdot \frac{|g(n)|}{|h(n)|} = 0 \cdot 0 = 0$$
■

Theorem If $\alpha < \beta$ then $x^\alpha \prec x^\beta$

Proof Left as an exercise to reader. ■

Theorem $f(n) \prec g(n) \iff \frac{1}{f(n)} \succ \frac{1}{g(n)}$

Proof Left as an exercise to reader. ■

Theorem $e^{f(n)} \prec e^{g(n)} \iff \limsup_{n \rightarrow \infty} (f(n) - g(n)) = -\infty$

Proof Using the definition, we will get:

$$\begin{aligned}
e^{f(n)} &\prec e^{g(n)} \\
&\iff \limsup_{n \rightarrow \infty} \frac{e^{f(n)}}{e^{g(n)}} = 0 \\
&\iff \limsup_{n \rightarrow \infty} e^{\ln\left(\frac{e^{f(n)}}{e^{g(n)}}\right)} = 0 \\
&\iff \limsup_{n \rightarrow \infty} e^{f(n)-g(n)} = 0 \\
&\iff \limsup_{n \rightarrow \infty} f(n) - g(n) = -\infty
\end{aligned}$$

Theorem Let $f(x) = k * g(x)$ for some real $k \neq 0$, then $f(x) \asymp g(x)$

Proof Left as an exercise to reader. ■

Theorem Let $f(x) \succ g(x)$ and $f(x) \asymp h(x)$ then $f(x) \asymp h(x) + g(x)$

Proof Left as an exercise to reader. ■

This previous last two theorems tells us that the equivalence induced by \asymp **Asymptotic Dominance** is invariant under scaling and translation by dominated functions. That is, in a sum of functions, we only need to consider the most dominant function when comparing asymptotic dominance. This allows us to talk about a group of functions using a single 'representative' function.

\asymp Representative Function wrt asymptotic dominance

The representative function of a set of asymptotically equivalent functions is the simplest form of the equivalence class, defined as:

- A function with a leading coefficient of 1,
- No additive subdominant terms (i.e., no terms in addition that are asymptotically dominated by others in the class).

For example x^2 is the representative function of the equivalence class of quadratic functions, $ax^2 + bx + c$.

Theorem If $f(x), g(x)$ are the representative functions of the equivalence class F, G ; then show that $f(x) \succ g(x) \iff \hat{f}(x) \succ \hat{g}(x)$ for all $\hat{f} \in F$ and $\hat{g} \in G$

Proof (\implies) Using the fact $f, \hat{f} \in F$, we get $f(x) \asymp \hat{f}(x) \iff \lim_{x \rightarrow \infty} \frac{f(x)}{\hat{f}(x)} = c_1$ where $c_1 \neq 0$ is a constant.

Similarly, $g(x) \asymp \hat{g}(x) \iff \lim_{x \rightarrow \infty} \frac{g(x)}{\hat{g}(x)} = c_2$ where $c_2 \neq 0$ is a constant.

Using $f(x) \succ g(x) \iff \lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = 0$

Thus,

$$\begin{aligned}
& \limsup_{x \rightarrow \infty} \frac{|\hat{g}(x)|}{|\hat{f}(x)|} \\
&= \limsup_{x \rightarrow \infty} \frac{|\hat{g}(x)|}{|g(x)|} \cdot \frac{|g(x)|}{|f(x)|} \cdot \frac{|f(x)|}{|\hat{f}(x)|} \\
&= \frac{1}{c_2} \cdot 0 \cdot c_1 \\
&= 0 \\
&\Rightarrow \hat{f}(x) \succ \hat{g}(x)
\end{aligned}$$

(\Leftarrow) $f \in F$ and $g \in g$ implies $f(x) \succ g(x)$ as this is true for all functions in the equivalence classes. ■

x An Hierarchy of Common Functions

Prove that:

$$1 \prec \log(\log(n)) \prec \log(n) \prec n^\epsilon \prec n^c \prec n^{\log(n)} \prec c^n \prec n! \prec n^n \prec c^{c^n}$$

§10.2.3. Big Oh notation

These topics seem disconnected, right? Let's fix that.

÷ Big Oh (from Big Ell)

$$f(x) = o(g(x))(n \rightarrow \infty) \iff \exists k, n_0 \text{ s.t. } \forall n \geq n_0 f(n) = k\mathcal{L}(g(n))$$

÷ Big Oh (from hierarchy)

$$f(n) = o(g(n))(n \rightarrow \infty) \iff f(n) \preceq g(n)$$

Theorem The above definitions are equivalent.

Proof Lemma: Both the definitions are equivalent to $\limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} < \infty$.

From the first definition, there exists an k, n_0 such that $\forall n_0 \leq n$:

$$\begin{aligned}
f(n) &= k\mathcal{L}(g(n)) \\
&\iff |f(n)| \leq k |g(n)| \\
&\iff \frac{|f(n)|}{|g(n)|} \leq (k)
\end{aligned}$$

As $n_0 < \infty$ and $k < \infty$; thus,

$$\limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} < \infty$$

The equivalence from the second definition follows from the definition of asymptotic dominance. ■

We also allow this lemma to also be a definition of o .

÷ Big Oh (limit)

$$f(n) = o(g(n))(n \rightarrow \infty) \iff \limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} < \infty.$$

¹²We could just say this part was trivial.

We also give the classical definition.

÷ Big Oh (Classical)

$$f(n) = o(g(n))(n \rightarrow \infty) \iff \exists c, n_0 \text{ s.t. } \forall n \geq n_0, |f(n)| \leq c \cdot |g(n)|$$

Theorem The above definitions are equivalent.

Proof Left as exercise. ■

While we have used \limsup here, it is only to deal with the cases where the limit is not guaranteed to exist. We can use \lim in place for all the definitions till here in most cases we will be concerned with.

In the family of big oh, we also have some more notations one must be aware of. This is called the Bachmann–Landau family, with contributions from Hardy and Knuth.

÷ Bachman-Landau Notation

Notation	Name	Classical Definition	Limit Definition
$f(n) = \Theta(g(n))$	Big Theta	$\exists k_1, k_2, n \text{ s.t. } \forall n > n_0, k_1 g(n) \leq f(n) \leq k_2 g(n)$	$f(n) \asymp g(n)$
$f(n) = \Omega(g(n))$	Big Omega	$\exists k, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \geq kg(n)$	$\liminf_{n \rightarrow \infty} \frac{ f(n) }{g(n)} = 0$
$f(n) \sim g(n)$	Total Asymptotic Equivalence (different for asymptotic equivalence defined using \asymp .)		$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$
$f(n) = o(g(n))$	Small Oh	$\exists k, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \leq kg(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \omega(g(n))$	Small Omega	$\exists k, n_0 \text{ s.t. } \forall n \geq n_0, f(n) > kg(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

Theorem If $f \in F$ equivalence class and \hat{f} is the representative of the class, then $f(x) = \Theta(\hat{f}(x))$

Proof Follows trivially from definition. ■

Here is an example to illustrate the difference between the notations.

- $3n^2 - 100n + 6 = o(n^2)$, because I choose $c = 3, n_0 = \frac{6}{100}$;
- $3n^2 - 100n + 6 = o(n^3)$, because I choose $c = 1, n_0 = \frac{7}{100}$;
- $3n^2 - 100n + 6 \neq o(n)$, because for any c I can take $n > 100 + c$;
- $3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2, n_0 = 100$;
- $3n^2 - 100n + 6 \neq \Omega(n^3)$, because for an c , I can take $n > \frac{100}{c}$, if $c < 1$ and $n > c$ otherwise;
- $3n^2 - 100n + 6 = \Omega(n)$, because for any c , I can take $n_0 = 100c$;
- $3n^2 - 100n + 6 = \Theta(n^2)$, because both o and Ω apply;
- $3n^2 - 100n + 6 \neq \Theta(n^3)$, because Ω fails;
- $3n^2 - 100n + 6 \neq \Theta(n)$, because o fails.

Also note, $3n^2 - 100n + 6 \sim 3n^2$ by taking the limit.

Finally, note that o, ω are equal to o, Ω here as our limits exist.

x Big Oh Arithmetic

Prove the following:

1. $n^m = o(n^{m'}) \quad m \leq m'$
2. $o(f(n)) + o(g(n)) = o(|f(n)| + |g(n)|)$
3. $f(n) = o(f(n))$
4. $o(o(f(n))) = o(f(n))$
5. $co(f(n)) = o(f(n))$ where c is constant
6. $o(f(n))o(g(n)) = o(f(n)g(n)) = f(n)o(g(n)) = g(n)o(f(n))$
7. $o(f(n)^2) = o(f(n))^2$

§10.3. Asymptotic Mathematics

Notice that Hardy was involved in developing some of this notation. What has an analytical number theory got to do with a tool for algorithmic analysis?

We wrote $n \rightarrow \infty$ next to o in the definitions as we are consider the limits approaching ∞ . We can also define similarly for $n \rightarrow 0$ and $n \rightarrow k$. These definitions are more common in math than CS, as mathematicians care about behavior of functions at a lot of places other than ∞ . As this is a CS book, in absence of clarification, assume $n \rightarrow \infty$.

This allows us to write taylor series in big-oh notation, for example as $\sin(x) = x - \frac{x^3}{3!} + o(x^5)(x \rightarrow 0)$.

All this can be made useful in problems like:

x Sum of Power of Numbers

Comment on the asymptotic behavior of

$$f_{k(n)} = \sum_{i=1}^n i^k$$

where $k, n \in \mathbb{N}$.

Let's start with some small cases.

$$f_0(n) = \sum_{i=1}^n i^0 = \sum_{i=1}^n 1 = n = o(n)$$

$$f_1(n) = \sum_{i=1}^n i^1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} = o(n^2)$$

$$f_2(n) = \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = o(n^3)$$

This seems to indicate $f_{k(n)} = o(n^{k+1})$, but how do we prove this?

÷ Derivative with big Oh

We can define the derivative of f through this equation¹³

$$f(x+h) = f(x) + hf'(x) + o(h^2) \quad (h \rightarrow 0)$$

÷ Binoial Exapansion with Big Oh

$$(1+x)^n = 1 + nx + \binom{n}{2}x^2 + \dots + \binom{n}{k}x^k + o(x^{k+1}) \quad (n \rightarrow 0)$$

÷ Taylor Series with Big Oh

$$f(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + \dots + f^{(i)}(0)\frac{x^i}{i!} + \mathcal{O}(x^{i+1}) \quad (x \rightarrow 0)$$

This allows us to define $(x+h)^{k+1} = x^{k+1} + h(k+1)x^k + o(h^2)$, taking $h = 1$;

$$(x+1)^{k+1} = (x)^{k+1} + (k+1)x^k + o(1)$$

$$(x+1)^{k+1} - (x)^{k+1} = (k+1)x^k + o(1)$$

$$\sum_{x=0}^n [(x+1)^{k+1} - (x)^{k+1}] = \sum_{x=0}^n [(k+1)x^k + o(1)]$$

$$\sum_{i=1}^{n+1} [i^{k+1} - (i-1)^{k+1}] = (k+1) \sum_{x=1}^n x^k + o(n)$$

$$\frac{(n+1)^{k+1} - o(n)}{k+1} = \sum_{x=1}^n x^k$$

$$\sum_{x=1}^n x^k = \frac{n^{k+1} + (k+1)n^k + o(1) - o(n)}{k+1}$$

$$\sum_{x=1}^n x^k = o(n^{k+1})$$

We can also use this notion to represent some mathematics results. For example:

÷ Prime Number Theorem

Let $\pi(n)$ represent the number of primes less then n . Then:

$$\pi(n) \sim \frac{n}{\ln(n)}$$

¹³This only works if f has a strong or strict derivative. For most functions we care about, this is true (and equals their usual derivative). This note is included to clarify in case you have doubts, despite the fact we do not wish to go into the further technicalities of this.

÷ Stirling Approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + o\left(\frac{1}{n^2}\right)\right)$$

¹⁴ Also note,

$$\begin{aligned} n! &\sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \\ \Rightarrow n! &= \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right) \end{aligned}$$

÷ Binomial Coefficient

When k is fixed,

$$\binom{n}{k} \sim \frac{n^k}{k!} = \Theta(n^k)$$

When n, k are both large, we can use the Stirling Approximation to get:

$$\binom{n}{k} \sim \sqrt{\frac{n}{2\pi k(n-k)}} \left(\frac{n^n}{k^k (n-k)^{n-k}}\right)$$

One case we have suspiciously left out is when k is small.

x Small k binomials

We will restrict ourselves to $k = o(n)$ as other cases get messy rather quickly.

Notice

$$\binom{n}{k} = A(n, k) \frac{n^k}{k!}$$

where

$$A(n, k) := \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right)$$

Thus, to find the asymptotics, we just need to solve for A .

$$\ln(A(n, k)) = \sum_{i=1}^{k-1} \ln\left(1 - \frac{i}{n}\right)$$

Solve when

- (i) $k = o(\sqrt{n})$
- (ii) $k = c\sqrt{n}$ and c is known
- (iii) $k = o(n^{\frac{2}{3}})$
- (iv) $k = o(n^{\frac{3}{4}})$
- (v) $k = o(n)$

¹⁴More terms can be obtained by using the Bernolli numbers. We have gone for this form as the proof follows from probability theory and doesn't involve results from complex analysis.

We will solve the first 3 cases and leave the rest for you, the reader.

(i) Using \div Taylor Series with Big Oh on $\ln(1+x) = x + o(x^2)$, we will get:

$$\begin{aligned}
 \ln(A(n, k)) &= \sum_{i=1}^{k-1} \ln\left(1 - \frac{i}{n}\right) \\
 &= -\sum_{i=1}^{k-1} \left(\frac{i}{n} + \mathcal{O}\left(\frac{i}{n}\right)^2 \right) \quad \left(\frac{i}{n} \rightarrow 0 \right) \\
 &\sim -\frac{k^2}{2n} + \mathcal{O}(k^3 n^{-2}) \quad \left(\frac{k}{n} \rightarrow 0 \right) \quad \text{using the sum of consecutive terms} \\
 &= o(1) \quad (k, n \rightarrow \infty) \quad \text{using } k = o(\sqrt{n})
 \end{aligned}$$

This implies $A = e^{o(1)} = o(1) \Rightarrow \binom{n}{k} = o(1) \cdot \frac{n^k}{k!} = o\left(\frac{n^k}{k!}\right)$.

(ii) Here the c being specified gives us $A = e^{-\frac{c^2}{2}} \Rightarrow \binom{n}{k} = e^{-\frac{c^2}{2}} \frac{n^k}{k!}$.

Hint for (iii), (iv), (v): We were able to dissolve the $\mathcal{O}(k^3 n^{-2})$ term as if $k = o(\sqrt{n})$, the term would be $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$ and goes to zero, rapidly.

We can 'fix' this by having a slightly bigger $k = o\left(n^{\frac{2}{3}}\right)$. The same happens in (iv) as we take a more and more accurate Taylor expansion for $\ln\left(1 - \frac{i}{n}\right)$. Can we get some sort of a limiting case and solve for $k = o(n)$?

x Bootstrap(Erdos, Greene)

Find the asymptotics of f satisfying:

$$f(t)e^{f(t)} = t$$

as $t \rightarrow \infty$

The idea here is to begin with a weak bound and progressively make it stronger.

Here we start by simplifying the equation:

$$\ln(f(t)) + f(t) = \ln(t) \quad (t \rightarrow \infty)$$

We know that $f(x) \succ \ln(f(x))$, thus

$$\begin{aligned}
 \lim_{t \rightarrow \infty} \frac{\ln(f(t)) + f(t)}{\ln(t)} &= 1 \\
 \Rightarrow \lim_{t \rightarrow \infty} \frac{f(t)}{\ln(t)} &= 1 \\
 \Rightarrow f(t) &= \Theta(\ln(t))
 \end{aligned}$$

Making this substitution gives us,

$$f(t) = \ln(t) - \Theta(\ln(\ln(t)))$$

Now, making this substitution gives us,

$$\begin{aligned}
& \ln(\ln(t) - \Theta(\ln(\ln(t)))) + f(t) = \ln(t) \\
& \ln\left(\ln(t) \left(1 - \Theta\left(\frac{\ln(\ln(t))}{\ln(t)}\right)\right)\right) + f(t) = \ln(t) \\
& \ln(\ln(t)) + \ln\left(1 - \Theta\left(\frac{\ln(\ln(t))}{\ln(t)}\right)\right) + f(t) = \ln(t) \\
& f(t) = \ln(t) - \ln(\ln(t)) + \Theta\left(\frac{\ln(\ln(t))}{\ln(t)}\right)
\end{aligned}$$

And one can continue getting more and more terms by this process. We stop here as getting the logitherm now will be much harder.

x A Weird Condition

Prove that there are infinitely many integers n such that the sum of digits of n in base $p < n$ exceeds 2025 for every prime p .

This question is incredibly hard to answer through regular means of construction of contradiction.

We will try to set up an asymptotic bound on number of integers violating the condition and show that their density is less than 1, hence, showing that the condition is true for infinitely many numbers.

Let's say $x < N$ integers violate these condition for some large N . Notice,

$$\forall \text{ primes } p \leq n, \exists k \in \mathbb{N} \text{ s.t. } n^{\frac{1}{k+1}} < p \leq n^{\frac{1}{k}}$$

Thus, given a p , any number from $1..N$ has atmost $k + 1$ digits in base p . Thus, a violation can only occur if the sum of digits: $d_1, d_2, \dots, d_{k+1} < p$ is less than 2025.

We can (grossly) overcount the number of violaters using the sum:

$$x < \sum_k \sum_{\substack{N^{\frac{1}{k+1}} < p \leq N^{\frac{1}{k}} \\ p \text{ is prime}}} \binom{k + 2026}{2025}$$

To might be as good time as any to get the bounds on k .

$$\begin{aligned}
\pi\left(N^{\frac{1}{k_m}}\right) - \pi\left(N^{\frac{1}{k_m+1}}\right) &= 0 \\
\frac{k_m N^{\frac{1}{k_m}}}{\ln(n)} - \frac{(k_m + 1) N^{\frac{1}{k_m+1}}}{\ln(n)} &= 0 \\
k_m N^{\frac{1}{k_m}} &= (k_m + 1) N^{\frac{1}{k_m+1}} \\
N^{\frac{1}{k_m(k_m+1)}} &= 1 + \frac{1}{k_m} \\
\frac{1}{k_m(k_m + 1)} \ln(N) &= \ln\left(1 + \frac{1}{k_m}\right) \\
\frac{1}{k_m(k_m + 1)} \ln(N) &= \frac{1}{k_m} + O\left(\frac{1}{k_m^2}\right) \\
\frac{1}{k_m + 1} \ln(N) &= 1 + O\left(\frac{1}{k}\right) \\
\ln(N) &= k + O(1) + O\left(\frac{1}{k_m}\right) \\
k_m &= \Theta(\ln(N))
\end{aligned}$$

Notice, for $k = 1$,

$$\sum_{\substack{\sqrt{N} < p \leq N \\ p \text{ is prime}}} \binom{2027}{2025} = \mathcal{O}(\pi(N)) < \frac{N}{3}$$

And for $k > 1$,

$$\begin{aligned}
&\sum_{k>1}^{k_m} \sum_{\substack{N^{\frac{1}{k+1}} < p \leq N^{\frac{1}{k}} \\ p \text{ is prime}}} \binom{k + 2026}{2025} \\
&= \sum_{k>1}^{k_m} \sqrt{N} \Theta(k^{2025}) \\
&= \sqrt{N} \Theta(\ln(N)^{2025}) \Theta(\ln(n)) \\
&= \sqrt{N} \Theta(\ln(N)^{2026}) \\
&< \frac{N}{3}
\end{aligned}$$

Thus, $x < \frac{2N}{3}$ for some n . Thus, atleast $\frac{N}{3}$ integers exist satisfying our property for some large N .

§10.4. Analysis of Algorithms

With the (damn scary) math out of the way, we will now move to algorithms and the real reason we are concerned with asymptotics. Take a deep breath, relax; the scary part is beyond us.

Before we get to designing and analyzing algorithms, let's pause and briefly question what 'algorithm' actually means. To quote Hannah Fry,

 Hannah Fry

It's a term that, although used frequently, routinely fails to convey much actual information. This is partly because the word itself is quite vague. Officially, it is defined as follows:

algorithm (noun): A step-by-step procedure for solving a problem or accomplishing some end especially by a computer.

An algorithm is simply a series of logical instructions that show, from start to finish, how to accomplish a task. By this broad definition, a cake recipe counts as an algorithm. So does a list of directions you might give to a lost stranger. IKEA manuals, YouTube troubleshooting videos, even self-help books – in theory, any self-contained list of instructions for achieving a specific, defined objective could be described as an algorithm. But that's not quite how the term is used. Usually, algorithms refer to something a little more specific. They still boil down to a list of step-by-step instructions, but these algorithms are almost always mathematical objects. They take a sequence of mathematical operations – using equations, arithmetic, algebra, calculus, logic and probability – and translate them into computer code. They are fed with data from the real world, given an objective and set to work crunching through the calculations to achieve their aim. They are what makes computer science an actual science, and in the process have fuelled many of the most miraculous modern achievements made by machines.

We will take Prof. Fry's definition as the gospel as trying to go into more details will open up questions which are more philosophical than we wish to be here.

§10.4.1. Sorting

A classic example in algorithm design is sorting. We have already seen some sorting algorithms, but here we will try to do an complete analysis of them.

There are two clear naive algorithms one can think about:

- We can repeatedly move through the list element by element, comparing the current element with the one after it, swapping their values if needed. Do these passes through the list are repeated until no swaps have to be performed in a pass, meaning that the list has become fully sorted. This is called Bubble Sort.
- We can take the first element as sorted list and then keep inserting the successive elements in a way that maintains the sorted property. This is called Insertion sort.

We can implement these ideas as:

```
bubble [] = ([], False)
bubble [x] = ([x], False)
bubble (x:y:ys)
  | x > y = (y : fst (bubble (x:ys)), True)
  | otherwise = (x : fst (bubble (y:ys)), snd (bubble (y:ys)))
bubblesort xs = let (a,b) = bubble xs in if b then bubblesort a else a

insert x [] = [x]
insert y (x:xs) = if x > y then y:x:xs else x : (insert y xs)

insertSort :: Ord a => [a] -> [a]
insertSort = foldr insert []
```

Bubble Sort is basically keeping the last element to its place every pass. So the first pass takes n comparisons and in the worst case, $n - 1$ switches. Similarly, the second pass takes n comparisons and at most $n - 2$ switches.

In the worst case, we will need to make n passes (the last one to check if the list is sorted). Thus, n^2 comparisons and $\frac{(n-1)n}{2}$ switches.

This is a very weird way to express the speed of the algorithm but the time to compare and switch two elements is different. So what do we do? Simple, use the fact that they are constant and take them as $\mathcal{O}(1)$. This assumption also makes our analysis easier by letting us remove some small factors and focus on large improvements.

Thus, bubblesort is $\mathcal{O}(n^2)$.

Similarly, `insert` takes $\mathcal{O}(k)$ time in the worst case to insert an element in a list of length k . Thus, `insertSort` takes $\mathcal{O}(1) + \dots + \mathcal{O}(n) = \mathcal{O}(n)$ time. (notice, we didn't talk about comparisons and switches as the big Oh allows us to think more simplistically).

The algorithms for sorting we saw in chapter 7 were merge sort and quick sort which worked as follows:

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

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

We had shown, through an cumbersome computation, that `mergesort` takes less than $n \log(n) + 3n + \frac{1}{2} \log(n) + \frac{1}{2}$ comparisons. Doing so with switches seems like a much sadder condition to be in.

But with the power of Big-Oh on our side, we just assume the worst case where every comparison leads to a switch and get the number of operations as $\mathcal{O}(n \log(n))$.

Similarly, we can use the results from there, to show that the worst case for `quicksort` is $\mathcal{O}(n^2)$ operations, but in the average case, it only takes $\mathcal{O}(n \log(n))$ operations.

But can we do better? Depends on how much you know about the elements of the list.

Say you have a bunch of spaghetti and you want to sort them. Well, just level them across the table. Then move your hand from top towards the tops of the spaghetti till your hand touches the tallest spaghetti, pull it out and continue. This is a $\mathcal{O}(n)$ algorithm. It even has a name Spaghetti Sort.

But let's say I have to sort a bunch of spaghetti sauces according to how much a distinguished person likes them and I can only ask them to compare two sauces at a time. In this case, we can prove that we can't do better than $\mathcal{O}(n \log(n))$.

Theorem An Sorting algorithm when we can't access the items' values directly: only compare two items and find out which is bigger or smaller can't be better than $\mathcal{O}(n \log n)$ in the worst case.

Proof In an n element list, knowing that $a_1 > a_2$ reduces the possible sortings from $n!$ to $\frac{n!}{2}$.

Similarly, knowing any more comparisons will reduce the number of possible orders by 2.

Thus, we need atleast $\lceil \log(n!) \rceil$ comparisons.

We claim $\lceil \log(n!) \rceil = \mathcal{O}(n \log(n))$. This is true as:

$$\begin{aligned} & \lceil \log(n!) \rceil \\ &= \lceil \log(n^n) \rceil \quad \text{as } n^n > n! \\ &= \lceil n \log(n) \rceil \\ &= \mathcal{O}(n \log(n)) \quad \text{as } f(x) - 1 \leq \lceil f(x) \rceil \leq f(x) + 1 \end{aligned}$$

■

This bound holds for randomized algorithms as well (like say a quickSort picking random pivots). The theorem is that no randomized algorithm can do better than $\mathcal{O}(n \log n)$ operations in the average case.

While proving so formally will require us to define some terms like probability distribution and bit tapes, here is the basic idea.

Given a list with infinite random numbers, our randomized algorithm uses these to do it's randomization. Thus, given this list, we have a normal non-random algorithm. Thus, we can now find the expectation and get that it has the same lower bound of $\mathcal{O}(n \log n)$.

Such proofs help us make sure we have achieved optimality and make sure people don't waste time looking for faster algorithms. However, we just sorted spaghetti in $\mathcal{O}(n)$ time, how?

The answer 'that was a stupid algorithm' is plain wrong as our proof never relied on dumbness or smartness. So what is at play?

As stated above, if we know more about what we are sorting, we could do better. For example, if they were spaghetti...

Given that most sorting is of integers and strings, why would you ever so restrict yourself as to only use comparisons? You can do much more with these objects that. You can add them, you can multiply them, you can count with them!

While it may seem unintuitive, but we can use even these operations for sorting! We will talk mostly about integer sorting algorithms but faster algorithms for strings also exists (which we will mention).

§10.4.1.1. Counting Sort

Let's say we want to sort a range of $0 - k$ with some number of repeats. So what do we do? We can do this in $\mathcal{O}(nk)$ time.¹⁵

Counting Sort

```
countingSort :: Int -> [Int] -> [Int]
countingSort k lis = concat (go k lis []) where
  go (-1) _ ans = ans
  go k lis ans = go (k-1) lis (filter (==k) lis : ans)
```

For a fixed k , this is already an $\mathcal{O}(n)$ algorithm; but it is easy to observe, that in practice this would behave $\mathcal{O}(n^2)$ -ish.

And secondly, if we want to fix a large k , it better be `maxBound :: Int` atleast. That makes the algorithm insanely slow while still being $\mathcal{O}(n)$.

So why talk about it? Because it is a subroutine to

¹⁵If we had an array, we could do this in $\mathcal{O}(n + k)$ time as we have random access.

§10.4.1.2. Radix Sort

We begin by first making modifications to `countingSortWithKey :: Int → [(Int, Int)] → [(Int, Int)]` which will sort based on the key value of the pairs (the second value of the pair) keeping the list otherwise stable (we don't change the order from that prescribed by original list if key value is same).

A Counting Sort With Keys

```
countingSortWithKey :: Int → [(Int, Int)] → [(Int, Int)]
countingSortWithKey k lis = concat (go k lis []) where
  go (-1) _ ans = ans
  go k lis ans = go (k-1) lis (filter (\y → k == snd y) lis : ans) -- The
single change!
```

≡ Radix Sort

Radix sort is a sorting algorithm which sorts a list of integers digit by digit, starting from the least significant digit; maintaining stability in subsequent sorts.

For example, if we had to sort:

853, 872, 265, 238, 199, 772, 584, 204, 480, 173,
499, 349, 308, 314, 317, 186, 825, 398, 899, 161

By the described process, We would first sort using the one's digit.

480, 161, 872, 772, 853, 173, 584, 204, 314, 265,
825, 186, 317, 238, 308, 398, 199, 499, 349, 899

We will now sort using the ten's digit, remember, we need to be stable that is for numbers that are tied on the middle digit, keep them in the current order.

204, 308, 314, 317, 825, 238, 349, 853, 161, 265,
872, 772, 173, 480, 584, 186, 398, 199, 499, 899

Finally, we will sort using the hundred's number.

161, 173, 186, 199, 204, 238, 265, 308, 314, 317,
349, 398, 480, 499, 584, 772, 825, 853, 872, 899

X Proof of Correctness

Show that Radix Sort correctly sorts an input list of n integers via induction of the length of longest number in the list.

Hint: You might want an induction hypothesis which looks more like the process. Something along the lines that the last j places are sorted in j th pass.

So how do we quickly sort the numbers by the last places? Use it as a key and use `countingSortWithKey`. This would look like:

```

digit pos y = (y `mod` (10 ^ (pos + 1))) `div` (10 ^ pos)

radixSort :: Int → [Int] → [Int]
radixSort maxLength lis = go 0 lis where
  go pos lis = if pos == maxLength then lis else go (pos + 1) newLis where
    key = map (digit pos) lis
    lisWithKey = zip lis key
    newLis = map fst $ countingSortWithKey 9 lisWithKey

```

This is a rather nice implementation, of what can easily be a very cumbersome code. A lot of helpers are created for more clarity instead of going for a more concise but impenetratable version.

Let's now compute the time complexity. We will take the number of digits in the base system to be d , the longest number be l long and the list having n numbers.

Thus, every pass through the list takes:

- $\Theta(n)$ time to get the digits.
- $\Theta(n)$ time to zip the key and numbers up.
- $\Theta(nd)$ time to counting sort with the given key.
- $\Theta(n)$ time to map `fst`.

This will be a total of $\Theta(nd)$ time. We will make l passes through the list.

Thus, the total time complexity is $\Theta(ndl)^{16}$. Setting $d = 9$, we would have $\Theta(n \log_d(M)) = \Theta(n \log(M))$ which would make it asymptotically faster for any list with the largest number less than number of elements.

x Complete Radix Sort

Modify the radix sort algorithm to make functions:

```

radixSortWithBase :: Int → Int [Int] → [Int]
-- Radix sort with some other base.
radixSortPack :: [Int] → [Int]
-- The packaged radixSort which just takes a list of ints and sorts
it, hiding the bells and whistles.

```

§10.4.1.3. Survey of Sorting Algorithms

In this survey, we will consider the optimal data structures. Design and Analysis of (some of) them is dicussed in ch11.

We will let $w = \log(M)$

Published	Algorithm / Authors	Data Structure	Complexity
Since Antiquity	Merge Sort	List	$\mathcal{O}(n \log(n))$
Since Antiquity	Radix Sort	Arrey, List	$\mathcal{O}\left(n \frac{w}{\log(n)}\right)$ and for list, $\mathcal{O}(nw)$
1974	van Emde Boas	van Emde Boas Tree	$\mathcal{O}\left(n \log\left(\frac{w}{\log n}\right)\right)$
1983	Kirkpatrick, Reisch	Trie	$\mathcal{O}\left(n + \frac{w}{\log(n)}\right)$

¹⁶Again, as the time complexity for counting sort is different for arrays, the complexity of radix sort would also change.

1995	Andersson, Hagerup, Nils-son, Raman (called Signature Sort)	Compressed Trie	$\mathcal{O}(n \log \log n)$ for $\log^{2+\varepsilon}(w) > n$
2002	Han, Throup	Too Weird	$\mathcal{O}(n \sqrt{\log \log n})$ for some nice bound on w) ¹⁷

An open question is if it is possible to do sorting in $\mathcal{O}(n)$. For example, if $w = \Omega(\log(n)) \Rightarrow$ Radix Sort is $\mathcal{O}(n)$.

What about smaller w ? This was given by Andersson et. al. where $\mathcal{O}(n)$ is achieved for $w = \mathcal{O}(n^{\frac{1}{2}-\varepsilon})$.¹⁸

Belazzougui et. al. in 2014 gave a way to sort in $\mathcal{O}(n)$ for $w = \Omega(\log^2(n) \log \log n)$. Their algorithm, called Packed Sort, works normally close to $\mathcal{O}(n \log n)$ but in certain cases becomes much faster.

A general proof or a single algorithm across w is not yet known and not much progress has been made in the last decade. Same is true for randomized algorithms like quick Sort; while results are slightly better their, progress has slowed down considerably.

§10.5. RAM Model and Asymptotic Analysis

A change in our approach in this chapter is the fact that we never told you what time algorithms took to run for us.

Radix sort was quite fast for whenever I ran it in my GHCI, it performed better than merge sort; but you needed neither my computer times nor took my word for it.

The thing is computers become more powerful all the time. An empirical relation which has been true upto recently is:

Moore's Law

The number of transistors in an integrated circuit (IC) doubles about every two years.¹⁹

So sometime in future, a mergesort on your computer would beat radix sort on mine. I need to argue that the radix sort will beat merge sort on your device; without seeing your device or the computer's processing paradigm then. Furthermore, algorithms almost always are faster in a low-level language like C or Assembly; and an algorithm is almost always slower in a high-level language like JavaScript or Python. Even worse, runtimes depend on how the compiler is optimizing your code, yes, some compilers do that.

With so many considerations, evaluating algorithms is quite hard.

This is quite a task and many methods have been proposed for a machine independent, language independent algorithm design. The first is to use a hypothetical machine called Random Access Machine or RAM (not to be confused with Random Access Memory which is a part of modern computer architecture).

¹⁷The reason we don't describe Han-Throup Algorithm well as none of us are that very intrested in sorting algorithms and hence, don't have the level of knowledge of tree structures and algorithms needed to do a description of this justice.

¹⁸One can also see the complexity when $\log^{2+\varepsilon} w > n$ in the table. The middle cases are where a complex complexity (pun intended) form with w and n can be obtained.

¹⁹

- It is an observation, not a law.
- It is argued to be slowing down, by some groups.

⚡ Random Access Machine

Under this model of computation, we are confronted with a computer where:

- Each simple operation (+, *, -, =, `if` and calling a function) takes exactly one time step.
- The called functions are not considered simple operations. Instead, they are the composition of many single-step operations. For example, if an algorithm calls `sort`, it should certainly take more than 1 time step.
- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Under the RAM model, we measure run time by counting up the number of steps an algorithm takes on a given problem instance.

The RAM is a simple model of how computers perform. Perhaps too simple. After all, multiplying two numbers takes more time than adding two numbers on most processors (and also is basically a subroutine as we will see at the end of this chapter); violating the first assumption of the model.

The presence of multiple processors and threads, which fancy compilers use, may as well violate the second assumption as we can be running the subroutine and routine at once.

And certainly memory access times differ greatly depending on whether data sits in cache or on the disk. Finally, we don't have ∞ memory.

This makes the model wrong on every single front. But here is a thing, it is still a useful model.

Consider the model that light travels in straight lines. From a physics standpoint, we know this is fundamentally incorrect. Light exhibits wave properties, can be bent by gravity, and behaves according to complex electromagnetic principles.

But when designing the lighting for a theater stage, the straight-line model of light is not only sufficient but essential. A lighting designer doesn't need to consider quantum mechanics or gravitational lensing when positioning spotlights; they simply need to predict where shadows will fall and how bright different areas will be. The straight-line model makes these calculations trivial and intuitive.

However, when that same lighting designer needs to design a laser show with mirrors, they might need the more sophisticated model that accounts for reflection and refraction. And if they were designing equipment for their university's astronomy lab (explains the need for part-time working in theater) studying distant galaxies, they'd need to consider how gravity bends light around massive objects. Each level of complexity serves its purpose within the appropriate domain.

This is the case with RAM as well. We use RAM machines to make our lives easier and as they are true when dealing with large inputs. The reason is that when dealing with large inputs, the cache-disk time, the addition-multiplication time etc doesn't matter. This is also why we normally use RAM along with asymptotic analysis; as we don't care if we can do better in small cases; we want to do better in large ones.

For dealing with memory, concurrency, parallelism, cache-disk etc other models of computation are created²⁰. You will possibly study them in your wider career.

§10.5.1. What Big O doesn't want you to know?

\mathcal{O} sweeps the constants under the rug. This works when the rug is large and heavy while the constants are mere dust specks.

²⁰RAM is a model in the class of models called Register Machines which are equivalent to Turing Machines.

But if, the constants are large: well then $10^{10^{100}} n$ is worse than n^2 for all values we could care about, irrespective of the fact the former is $\mathcal{O}(n)$ while the latter is $\mathcal{O}(n^2)$. This leads to something called Galactic Algorithms.

≡ Galactic Algorithms

A galactic algorithm is one with an optimal theoretical asymptotic performance, but which is never used in practice. Typical reasons are that the performance gains only appear for problems that are so large they never occur, or the algorithm's complexity outweighs a relatively small gain in performance. Galactic algorithms were so named by Richard Lipton and Ken Regan, because they will never be used on any data sets on Earth.

This is of matter to us as at the end of all this theoretical work, we wish to use our algorithms or at least know when to switch algorithms. For example, a lot of languages use insertion sort to sort till $n = 5$ or 6 before going to merge sort.

Hence, sometimes one really does the grulling analysis with the constants we saw in chapter 7.

This may make it seem useless. But the fact is, an algorithm, even if impractical, may create new techniques that may eventually be used to create practical algorithms.

Also, an impractical algorithm can still demonstrate that conjectured bounds can be achieved, or that proposed bounds are wrong, and hence advance the theory of algorithms.

Richard J. Lipton, Kenneth W. Regan

This alone could be important and often is a great reason for finding such algorithms. For example, if tomorrow there were a discovery that showed there is a factoring algorithm with a huge but provably polynomial time bound, that would change our beliefs about factoring. The algorithm might never be used, but would certainly shape the future research into factoring.

§10.6. An Informal Survey of Multiplication Algorithms

The word Algorithm originates from French where it was the mistranslated name of the 9th Century Arabic scholar Al-Khwarizmi, who was born in present day Uzbekistan, who studied and worked in Baghdad. His text on multiplying indo-arabic numerals travelled to Europe and his name was mis translated to “Algorisme” which later evolved into algorithm. While we will see other algorithms of the ancients in the exercise, let's end the main text with the OG multiplication. We will consider the multiplication of two n digit numbers, given it takes a single operation to solve for $n = 1$ (base cases). Our model of computation will be RAM but without the assumptions on addition and multiplication.

We assume additions of single digit takes $\mathcal{O}(1)$ (constant) time, and hence, adding a m, n digit number takes $\mathcal{O}(\min(m, n))$ time. This is realized by the school book carry method of addition and is optimal²¹. Similar proof holds for subtraction.

²¹The proof for the optimality is much harder. It was given by Emil Jerabek in 2023 using a technique we will see in ch 11 called Amortization

⚡ Addition and Subtraction of two numbers

```
-- Note, we are taking the numbers in reverse the usual order. That is Ones
-- → Tens → Hundreds ...
addNum num1 num2 = addNumCarry num1 num2 0 where
  -- Both numbers exhausted : if carry is 0, end otherwise append the carry.
  addNumCarry [] [] k = if k == 0 then [] else [k]

  -- First number exhausted: add carry to remaining digits of second number
  addNumCarry [] (y:ys) k =
    let (a,b) = (k+y) `divMod` 10
    in if a == 0 then b:ys else a:b:ys

  -- Second number exhausted: add carry to remaining digits of first number
  addNumCarry (x:xs) [] k =
    let (a,b) = (k+x) `divMod` 10
    in if a == 0 then b:xs else a:b:xs

  -- Main case: add corresponding digits plus carry, propagate new carry
  addNumCarry (x:xs) (y:ys) k =
    b : addNumCarry xs ys a
    where (a,b) = (x+y+k) `divMod` 10 -- a is new carry, b is digit result

subNum :: [Int] → [Int] → [Int]
subNum num1 num2 = subNumBorrow num1 num2 0 where
  subNumBorrow [] [] b = if b == 0 then [] else error "Result would be
negative"
  subNumBorrow [] (y:ys) b = error "Result would be negative"
  subNumBorrow (x:xs) [] b =
    let diff = x - b
    in if diff < 0
       then 9 : subNumBorrow xs [] 1 -- borrow from next digit
       else if diff == 0 && xs == [] then [] -- remove leading zeros
       else diff : xs
  subNumBorrow (x:xs) (y:ys) b =
    let diff = x - y - b
    in if diff < 0
       then (diff + 10) : subNumBorrow xs ys 1 -- borrow from next digit
       else diff : subNumBorrow xs ys 0
```

The naive way to multiply would be to define multiplication as repeated addition. Something of the sort `multiply 1 b = b` and the recurrence. `multiply a b = b + multiply (a-1) b`. For two n digit numbers, this will take $\mathcal{O}(10^n) \cdot \mathcal{O}(n) = \mathcal{O}(n10^n)$ operations. We normally use this to define single digit multiplications (as writing the table by hand is too cumbersome and in one digit case, it works just fine).

⚡ Singluar Digit Multiplication

```
mulDig :: Int → Int → [Int]
mulDig 0 _ = [0]
mulDig _ 0 = [0]
mulDig dig1 dig2 = [dig1] `addNum` mulDig dig1 (dig2-1)
```

An improvement in the multiplication algorithm, we are already familiar with is the one taught in school. This was also the algorithm Al-Khwarizmi found.

⚠ School Book Multiplication

```
-- Multiplying a number with a digit
mulNumDig :: [Int] -> Int -> [Int]
mulNumDig num dig = foldl1 (\a b -> a `addNum` (0:b)) (map (mulDig dig)
num)

schoolMul :: [Int] -> [Int] -> [Int]
schoolMul num1 num2 = foldl1 (\a b -> a `addNum` (0:b)) (map (mulNumDig
num2) num1)
```

The number of operations for this algorithm is $\mathcal{O}(n^2)$ as we multiply all the digits in the latter number with the digits in the former number and then add the results suitably one by one. That is in $32 * 45$, we will compute $32 * 5$ and $32 * 4$ and add them.

Doing a lot better than this took about a millenia, with the improvement coming from Anatoly Karatsuba in 1962. The idea used is the usual divide and conquer.

We divide the first number into $x = a * 10^{\frac{n}{2}} + b$ and the second number as $y = c * 10^{\frac{n}{2}} + d$. Thus,

$$\begin{aligned} xy &= (a * 10^{\frac{n}{2}} + b) * (c * 10^{\frac{n}{2}} + d) \\ &= ac * 10^n + bc * 10^{\frac{n}{2}} + ad * 10^{\frac{n}{2}} + bd \end{aligned}$$

⚠ Naive Divide and Conquer definition

```
divNcon :: [Int] -> [Int] -> [Int]
divNcon [x] num = mulNumDig num x
divNcon num [x] = mulNumDig num x
divNcon num1 num2 = let
  n = length num1
  n2 = n `div` 2
  (a,b) = splitAt n2 num1
  (c,d) = splitAt n2 num2
  ac = divNcon a c
  ad = divNcon a d
  bc = divNcon b c
  bd = divNcon b d
  n2zeros = replicate n2 0
  nzeros = replicate n 0
  in ac `addNum` (n2zeros ++ bc) `addNum` (n2zeros ++ ad) `addNum` (nzeros +
+ bd)
```

Let's say it takes $T(n)$ operations to multiply two n digit numbers. Thus, our problem of multiplying two n digit numbers can be reduced to multiplying two $\frac{n}{2}$ digit number 4 times.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) + \mathcal{O}(1) \\ \Rightarrow T\left(\frac{n}{2}\right) &= 4T\left(\frac{n}{4}\right) + \mathcal{O}\left(\frac{n}{2}\right) + \mathcal{O}(1) \\ \Rightarrow \dots \Rightarrow T(n) &= 4^{\log(n)} + \mathcal{O}(n)1 * \log(n) \\ \Rightarrow T(n) &= n^2 + \mathcal{O}(\log(n)) = \mathcal{O}(n^2) \end{aligned}$$

But we didn't do any better! All the effort seems to be in vain. Well, this is where Karatsuba's brilliance comes to play.

$$xy = ac * 10^n + bc * 10^{\frac{n}{2}} + ad * 10^{\frac{n}{2}} + bd$$

$$xy = ac * 10^n + (bc + ad) * 10^{\frac{n}{2}} + bd$$

We only need three terms. If we could somehow figure out $bc + ad$ without needing to find them both individually. The genius idea was:

$$(a + b)(c + d) = ac + bd + (bc + ad)$$

and we already have computed ac and bd . If we subtract them, we will have the third term. Notice, $a + b$, $c + d$ are at most an $\frac{n}{2} + 1$ digit numbers. Thus, we can now only make three smaller multiplications. We will claim $T(n + 1) = T(n) + \mathcal{O}(n)$ as we can divide the given $n + 1$ digit numbers into two parts, the most significant n digits and then the last digit and complete the computation.

This will give us,

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) + \mathcal{O}(1)$$

$$\Rightarrow T(n) = 3^{\log(n)} + \mathcal{O}(n) + \mathcal{O}(1) \log(n)$$

$$\Rightarrow T(n) = \mathcal{O}(n^{\log(3)}) \approx \mathcal{O}(n^{1.6})$$

Here is an Haskell implementation of the same

 Karatsuba Multiplication algortithm

```
karatsurba :: [Int] -> [Int] -> [Int]
karatsurba [x] num = mulNumDig num x
karatsurba num [x] = mulNumDig num x
karatsurba num1 num2 = let
    n = length num1
    n2 = n `div` 2
    (a,b) = splitAt n2 num1
    (c,d) = splitAt n2 num2
    ac = karatsurba a c
    bd = karatsurba b d
    abcd = karatsurba (a `addNum` b) (c `addNum` d)
    adPlusbc = (abcd `subNum` ac) `subNum` bd
    n2zeros = replicate n2 0
    nzeros = replicate n 0
    in ac `addNum` (n2zeros ++ adPlusbc) `addNum` (nzeros ++ bd)
```

This is a lot better. The next improvement came just an year later in 1963 by Tooom and Cook, making it $\mathcal{O}(n^{\log_3(5)})$. Here is what we belive their research process looked like:



If you are wondering, they showed that we can break the multiplication in five $\frac{n}{3}$ sized products. Actually, we can split in any number of parts we want. Karatsuba is Toom-2, the $\mathcal{O}(n^{\log_3(5)})$ algorithm is Toom-3. When split in some k parts, the complexity is $\mathcal{O}(n^E)$ where $E = \log_{k(2k-1)}$.

This can in theory do $\mathcal{O}(1)$ multiplication. As we have seen in the section about the dark secrets of big-oh, the constants will cause the problem. Doing an exact complexity analysis can allow us to compute the exact speed of growth of the constant of $\mathcal{O}(n^E)$ (hint: It is basically exponential).

x Toom-Cook

Making the required modifications to the haskell implementation of karatsurba, implement Toom-3 algorithm.

This leads us to the $\mathcal{O}(n \log(n) \log(\log(n)))$ Schönhage–Strassen algorithm (1971) which uses the Discrete Fast Fouries Transform algorithm described in chapter 8. The exact implementation can be found in the appendix, if you are morbidly curious regarding the same. In this paper, Arnold Schönhage and Volker Strassen also conjectured a lower bound of $\Omega(n \log(n))$.

This is about the end of multiplication algorithms I can hope to talk about with the material in this book. Also, the upcoming algorithms are ⚡ Galactic Algorithms which is why most implementations (even in the most complex mathematical computation software) use Karatsuba or Toom-3 till some size and then switch to Schönhage–Strassen. So everything here onwards are just fun facts.

The next leap came in 2007, when Martin Fürer improved the bound to $\mathcal{O}(n \log(n) 2^{\mathcal{O}(\log^*(n))})$ where the $\log^*(n)$ denotes the number of times we must take $\log(n)$ before we go below 1. This leap was made possible due to half-DFT's, lots of ring theory and complex analysis and certain results about primes of

form $p = 2^{2^k} + 1$ turning up true. This algorithm beats Schönhage–Strassen for integers with about 10^{19} digits.

The next improvement came by using number theory in place of complex analysis (bundled with other suitable changes) courtesy Anindya De, Piyush P Kurur, Chandan Saha and Ramprasad Saptharishi²² in 2008. Their algorithm beats this Fürer for numbers with about 10^{4796} digits.

In 2015, David Harvey, Joris van der Hoeven and Grégoire Lecerf gave a new algorithm which replaced the $\mathcal{O}(\log^*(n))$ in Fürer with $3 \log^*(n)$. The only issue is that this paper used certain unproven conjectures on Mersenne primes.

In 2015-16, in a series of two papers, Svyatoslav Covanov and Emmanuel Thomé first made a new algorithm with same complexity as Fürer and then, using unproven conjectures on Fermat Primes and generalizations, produced an algorithm where $\mathcal{O}(\log^*(n))$ is replaced with $2 \log^*(n)$.

Not to be defeated so easily, Harvey and Hoeven snapped back in 2018 with an algorithm which achieves the same complexity as Covanov and Thomé without any conjectures. They instead used Minkowski's theorem (which funnily was proven in 1889).

And to seal the deal, Harvey and Hoeven published the first $\mathcal{O}(n \log(n))$ multiplication algorithm in March 2019.

 David Harvey and Joris van der Hoeven

...our work is expected to be the end of the road for this problem, although we don't know yet how to prove this rigorously."

So well, can we do better is still an open question.

Anyways, for this paper, they shared the de Bruijn medal in 2022. With that, we have come full circle I guess.

²²Who was at the same institute (Chennai Mathematical Institute) as the authors when he made this algorithm. The other authors were from IIT Kanpur.

Advanced Data Structures

§ 11.1. Datatypes (One last time)

We have discussed how we can use datatypes to represent data in a cleaner way.

Towards the end of that chapter, we had implemented a binary tree and mentioned that some algorithms are implemented using binary trees. That seems to indicate that we might be interested in storing data using a data type for ease of working and then recovering the data? In this case, we call the datatype a data structure.

We will talk about some common data structures in this chapter.

§ 11.2. Stacks and Queues

Before moving to more complex data structures, we will talk about simpler data structures, which are degradingly called containers.

≡ Containers

A data structure that permits storage and retrieval of data items independent of content is called a container.

That is, the data structure has no interest in examining what we are storing and optimizing for that, it is purely a vessel for the data. This makes containers only interesting and distinguished by the insertion and retrieval order they support.

§ 11.2.1. Stack

≡ Stack

A container that supports insertion and retrieval in a first in last out fashion is called a stack.

It is as if we have a stack of books, we can add things to the top and remove from the top.

The insertion and retrieval in this case are called push and pop.

The term push and pop originate from Samelson and Bauer's paper on evaluating expressions (we will see it in a moment), where they called this container cellar.

A cellar is an underground storage vertical hole used to store wine casks. So to insert a new cask, we will need to push it in and to retrieve a new cask, we will need to pull it upward (which normally made an audible pop). We sometimes provide two more functions `peek` and `empty` which allow us to inspect the top element as well as check if the stack is empty.

We can implement this in Haskell as

```

module SafesStack where

data Stack a = Empty | Top a (Stack a) deriving Show

empty :: Stack a → Bool
empty Empty = True
empty _ = False

pop :: Stack a → Maybe (a, Stack a)
pop Empty = Nothing
pop (Top a s) = Just (a, s)

push :: a → Stack a → Stack a
push a s = Top a s

peek :: Stack a → Maybe a
peek Empty = Nothing
peek (Top a _) = Just a

```

and an unsafe version for easier use:

```

module Stack where

data Stack a = Empty | Top a (Stack a) deriving Show

empty :: Stack a → Bool
empty Empty = True
empty _ = False

pop :: Stack a → (a, Stack a)
pop Empty = error "Empty Stack"
pop (Top a s) = (a, s)

push :: a → Stack a → Stack a
push a s = Top a s

peek :: Stack a → Maybe a
peek Empty = error "Empty Stack"
peek (Top a _) = a

```

The official haskell version is unsafe and we will use the unsafe version throughout the section.

Stack from List

It is not hard to notice that the list we are familiar with are very similar to stacks. Can we implement stack using list?

```

data Stack a = Stack [a]

empty :: Stack a → Bool
pop :: Stack a → Maybe (a, Stack a)
push :: a → Stack a → Stack a
peek :: Stack a → Maybe a

```

Remember, the `data` keyword obscures the underlying structure when called as a `module`. This would make it impossible for someone to access the list, unless they are in the file defining the container itself. Anyways, so why is this rather simple structure any useful?

x Reverse Polish Notation

One of the first uses of computers was to compute arithmetic equations. The brackets were however a kludge to deal with.

$$3 * 4 + (6/3 - 5) - 8/2$$

is quite non trivial for computers to deal with. So what do we do about it?

One idea, proposed by Charles Hamblin was using a reversed version of Jan Łukasiewicz's bracket less notation. Łukasiewicz proposed writing arithmetic in operator first, inputs second way that is: $2 + 3$ is transformed into $+ \ 2 \ 3$. Similarly, the above expression is transformed into:

$$- \ + \ * \ 3 \ 4 \ - \ / \ 6 \ 3 \ 5 \ / \ 8 \ 2$$

The computation is easier to show by example than describe in words

$$\begin{aligned} & - \ + \ * \ 3 \ 4 \ - \ / \ 6 \ 3 \ 5 \ / \ 8 \ 2 \\ = & - \ + \ 12 \ - \ / \ 6 \ 3 \ 5 \ / \ 8 \ 2 \\ = & - \ + \ 12 \ - \ 2 \ 5 \ / \ 8 \ 2 \\ = & - \ + \ 12 \ - \ 3 \ / \ 8 \ 2 \\ = & - \ 9 \ / \ 8 \ 2 \\ = & - \ 9 \ 4 \\ = & 5 \end{aligned}$$

The idea of reverse polish notation is to keep the inputs first and the operator second: $2 + 3$ is transformed to $2 \ 3 \ +$. This transforms our expression to:

$$3 \ 4 \ * \ 6 \ 3 \ / \ 5 \ - \ + \ 8 \ 2 \ / \ -$$

Can you make a scheme to evaluate this? Can you make a computer evaluate this?

The idea we expect you to see is that we keep adding the numbers and operations to the stack. As soon as we see an operator, we perform the operation on the first two elements of the stack and push that to the stack. We continue doing so till we have read the input completely.

$$\begin{aligned} & 3 \ 4 \ * \ 6 \ 3 \ / \ 5 \ - \ + \ 8 \ 2 \ / \ - \\ = & 12 \ 6 \ 3 \ / \ 5 \ - \ + \ 8 \ 2 \ / \ - \\ = & 12 \ 6 \ 3 \ / \ 5 \ - \ + \ 8 \ 2 \ / \ - \\ = & 12 \ 2 \ 5 \ - \ + \ 8 \ 2 \ / \ - \\ = & 12 \ - \ 3 \ + \ 8 \ 2 \ / \ - \\ = & 9 \ 8 \ 2 \ / \ - \\ = & 9 \ 4 \ - \\ = & 5 \end{aligned}$$

The implementation would look like:

```

data Operator = Plus | Minus | Mul | Div deriving Show

evaluate :: [Either Integer Operator] → Integer
evaluate expression = go expression Empty
  where
    go [] acc = peek acc
    go (token:xs) acc = case token of
      Left num   → go xs (push num acc)
      Right op   →
        let (rhs, acc1) = pop acc
            (lhs, acc2) = pop acc1
        in case op of
          Plus   → go xs (push (lhs + rhs) acc2)
          Minus  → go xs (push (lhs - rhs) acc2)
          Mul    → go xs (push (lhs * rhs) acc2)
          Div    → go xs (push (lhs `div` rhs) acc2)

```

The time complexity is clearly $O(n)$. On my computer, I was able to evaluate a 300 thousand term expression in half a second.

x Ease of Use

It is rather cumbersome to type `[Left 3, Left 4, Right Plus, Left 5, Left 6, Right Plus, Right Mul]` in place of `"3 4 + 5 6 + *"`. Make a function `convert :: String → [Either Integer Operator]` to add some ease of use.

We will now give you as exercise to write a infix calculator (which can calculate arithmetic in the usual notation) using the below described algorithm.

x Samelson & Bauer's Sequential Formula Translation

Translate the following algorithm to Haskell. You will need to use two stacks, one of type `Stack Integer` and another of type `Stack Operator`. Also, parenthesis are also operators now.

- Read the expression from left to right. Let the token in consideration to be `x`
 - If `x` is an integer, push `x` to the Integer stack.
 - If `x` is an operator,
 - Evaluate operators from operator stack till either
 - Operator stack is empty
 - the top of the operator stack is an open parenthesis
 - The precedence of the operator at the top of operator stack is lower than `x`
 - Push `x` onto the operator stack
 - If `x` is an open parenthesis, push `x` onto the operator stack.
 - If `x` is a closed parenthesis,
 - Evaluate operators untill an open parenthesis is at the top of the operators stack.
 - Remove the open parenthesis.
- If the input string is empty, evaluate the remaining operators.

The precedence of operators is `/ = *` `>` `+ = -`. Note, equal precedence is not lower/higher precedence.

Evaluating an operator means popping two elements from the number stack, say a and b respectively and popping one element from the operator stack, say \oplus and returning $b \oplus a$ (a was popped first).

A question you might have is if stack is just a more constrained list, why not just use the list? Well, for haskell, at compile time, there is no real difference. Stack is a historically significant data structure and is sometimes makes the algorithm more clear (as with the above examples). We think of a list and stack, just by the words in a different manner and hence, it is just a thinking tool.

It is also rather similar to a lot of humane thinking we do.

x Mass of a Molecule

An organic molecule can be defined as a sequence of atoms and represented by a chemical formula consisting of letters denoting these atoms. E.g. letter H denotes atom of hydrogen, C denotes atom of carbon, O denotes atom of oxygen, formula CH3COOH represents the molecule acetic acid consisting of two atoms of carbon, four atoms of hydrogen and two atoms of oxygen.

To write formulas efficiently, letters can be grouped in parentheses, and groups may be nested. Consecutive identical atoms or groups can be compressed with a number, e.g. COOHHH = CO2H3, and CH(CO2H)(CO2H)(CO2H) = CH(CO2H)3. A number after a letter or group is always between 2 and 9. A mass of a molecule is the sum of masses of all atoms: H = 1, C = 12, O = 16.

Write function `mass :: String → Integer` to find the mass of a given organic molecule.

Examples

```
mass "CH3(COOH)" = 60 -- Acetic Acid aka Vinegar
mass "CH2(COOH)2" = 104 -- Malonic Acid, found in citrus fruits
mass "CH3C(CH3)CHCH2CH2C(CH3)CHCH2(OH)" = 154 -- Geraniol, the rose
smell
```

Hint : You want to use a `Stack (Either Integer Char)` and the only `Char` we are interested in are the parentheses.

§ 11.2.2. Queue

÷ Queue

A container that supports insertion and retrieval in a first in first out fashion is called a queue.

It is similar to a line at a ticket counter, you can only join at the back and exit (with a ticket) at the front.

The insertion and retrieval in this case are called enqueue and dequeue.

Unlike stack, it is somewhat harder to think of an efficient way to implement queues. A naive solution will be:

```
data NaiveQueue a = Queue [a] deriving Show

empty :: NaiveQueue a → Bool
empty (Queue lis) = null lis

enqueue :: a → NaiveQueue a → NaiveQueue a
enqueue a (Queue lis) = Queue (lis ++ [a])

dequeue :: NaiveQueue a → (a, NaiveQueue a)
dequeue (Queue []) = error "empty"
dequeue (Queue (x:xs)) = (x, Queue xs)

peek :: NaiveQueue a → a
peek (Queue []) = error "empty"
peek (Queue (x:xs)) = x
```


The issue is that our enqueue function takes $O(n)$ time where n is the number of elements already in the queue. This is a glaring inefficiency! Can we do any better?

```
data BatchQueue a = Queue [a] [a] deriving Show

empty :: BatchQueue a → Bool
empty (Queue f r) = null f && null r

enqueue :: a → BatchQueue a → BatchQueue a
enqueue a (Queue f r) = Queue f (a:r)

dequeue :: BatchQueue a → (a, BatchQueue a)
dequeue (Queue [] r) = dequeue (Queue (reverse r) [])
dequeue (Queue (f:fs) r) = (f, Queue fs r)

peek :: BatchQueue a → a
peek (Queue [] r) = peek (Queue (reverse r) [])
peek (Queue (f:fs) _) = f
```

We are storing two lists with one being the front of the queue and the other being the back. But one might, rightfully ask, what does this added complexity get us? We are using a `reverse` making the dequeue $O(n)$ in the worst case. That makes this no better, right?

Not really. In the last chapter, we had mentioned amortization. It's time to talk about it.

§ 11.2.2.1. Amortization

⌘ Ammortization

The action or process of gradually writing off the initial cost of an asset is called amortization.

Amortization is an accounting term which is generally used by firms to slowly write off expensive purchases. The concept is:

Say Prof. Wupendra Wulkarni buys a new (and expensive) laptop worth 20000M\$ to play chess on. On any 'work' related purchases, The University of Baker Street (TUBS) offers a payback upto 400M\$.

Instead of being honest and reporting the purchase at once, Wupendra can instead report a cost for renting a computer every month (from himself) which conveniently comes to be 400M\$. This scheme will allow him to slowly write off the purchase over 50 months. Note, Wupendra paid the price for the laptop at once, he is just doing the write off over a period of time.

This is called Ammortization in accounting. It is generally a valid practice and is often done to prevent showing a quatar with some purchases as a loss making quatar.

We have a similar concept in the study of data structures as well. Robert Tarjan in 1985 formalized the concept of Ammortized analysis in CS as:

⚡ Ammortized Analysis

Given a sequence of operations, we may wish to know the running time of the entire sequence, but not care about the running time of any individual operation. For instance, given a sequence of n operations, we may wish to bound the total running time of the sequence by $O(n)$ without insisting that each individual operation run in $O(1)$ time. We might be satisfied if a few operations run in $O(\log n)$ or even $O(n)$ time, provided the total cost of the sequence is only $O(n)$.

Formally, To prove an amortized bound, one defines the amortized cost of each operation and then proves that, for any sequence of operations, the total amortized cost of the operations is an upper bound on the total actual cost.

$$\sum_{i=1}^n a_i \geq \sum_{i=1}^n t_i$$

where t_i is the actual cost of operation i and a_i is the amortized.

To illustrate, we will prove that enqueue and dequeue are $O(1)$ amortized. We will prove this by induction.

Note, the actual costs of enqueue and dequeue are $O(1) = c_1$ and $O(k) = c_2 k$ where k is number of enqueue's before the last dequeue respectively. We will represent the chain of operations using e 's and d 's meaning enqueue and dequeue respectively.

Theorem enqueue and dequeue are performed in $c_0 = c_1 + c_2 = O(1)$ time amortized.

Proof B: For $n = 1$, the only operation we can make is e which is consistent with the claim as $c_1 \leq c_0$.

For $n = 2$, we can either do e, d or e, e . Note, this is also consistent with the claim as $c_1 + c_2 \leq 2c_0$ and $c_1 + c_1 \leq 2c_0$.

S: Assume that for any chain of operations of length $k < n$, it is bounded by $c_0 k$. All chains will be of the form

$$\underbrace{e \dots d}_k \quad \underbrace{e \dots e}_{n-k-1} \quad d$$

This will take time

$$\begin{aligned} t &\leq c_0 k + c_1(n - k - 1) + c_2(n - k - 1) \\ &\leq c_0 k + (c_1 + c_2)(n - k - 1) \\ &\leq c_0 k + (c_0)(n - k - 1) \\ &\leq c_0(n - 1) \\ &\leq c_0 n \end{aligned}$$

as required. ■

Another way to deal with amortization is using something called the “Piggy Bank method” which allows us to do such analysis in an easier way. In coming up with the above proof, we had to take care of what to choose as c_0 as well as deal with the induction that can be much more complicated for weirder data structures.

⌘ Piggy Bank Method

We will define this by example.

Let's say we pay 1\$ to perform an enqueue and deposit 1\$ in the piggy bank for a total cost of 2\$. Also, let's say that reversing a list costs 1\$ per element moved and 1\$ to start the process and dequeuing from the top of the queue just costs 1\$.

We can see that we can only reverse the number of elements we enqueue, hence, our total cost is n \$ at the end of a chain of n operation. That would make the amortized cost 1\$.

This method is often very useful for performing amortized analysis. The piggy bank is also called a potential function, since it is like potential energy that you can use later. The potential function is the amount of money in the bank.

In the case above, the potential is equal to the number of elements in the reverse list. Note that it is very important in this analysis to prove that the bank account doesn't go negative. Otherwise, if the bank account can slowly drift off to negative infinity, the whole proof breaks down.

We will see amortization again when dealing with more complicated data structures, and in that case the piggy bank method would serve us better.

§11.2.2.2. The True Queue

The problem still is that we are only promised $O(1)$ amortized, can we get a $O(1)$ worst case? The answer is yes, but it will take three lists.²³

The biggest barrier to all this is the reversing. What if we reverse incrementally? That would imply that we never want the front of the list to be empty. That would mean, waiting for it to be empty to do something which is not worth it...

What if instead, we have the rule that the front of the list is always as large as the rear? In this case, every enqueue or dequeue we will need to maintain this property. We will make this accumulator a suffix list which is, as the name suggests, a suffix of the front list. What we want is `length suffix = length front - length rear = drop (length rear) front`.

Whenever enqueue a new element to the rear, we drop the first element of the suffix if it is not empty or start the rotation process if it is empty. Similarly, whenever we dequeue a new element, we drop the first element of the suffix if it is not empty or start the rotation process if it is. The implementation looks like:

²³We are genuinely surprised how many Haskell (and general functional programming) sources make no mention of this further efficiency, given that it is known since 1995. This is also exactly how the Haskell's `Queue` package was implemented before its deprecation.

The queue equivalent now is `Sequence` which is a doubly linked list (list but both the front and back are accessible and recursive on in $O(1)$). Basically, it has the same relation to queue as list has with stack.

```

data TrueQueue a = Queue [a] [a] [a] deriving Show

empty :: TrueQueue a → Bool
empty (Queue f r s) = null f && null r

enqueue :: a → TrueQueue a → TrueQueue a
enqueue a (Queue f r s) = makeEq f (a:r) s

dequeue :: TrueQueue a → (a, TrueQueue a)
dequeue (Queue (f:fs) r s) = (f, makeEq fs r s)

peek :: TrueQueue a → a
peek (Queue (f:fs) _ _) = f

makeEq :: [a] → [a] → [a] → TrueQueue a
makeEq f r [] = let l = rotate f r [] in Queue l [] l
makeEq f r (s:ss) = Queue f r ss

rotate :: [a] → [a] → [a] → [a]
rotate [] (r:rs) acc = r:acc -- Front being empty would imply that the rear
only has a single element.
rotate (f:fs) (r:rs) acc = f : (rotate fs rs (r:acc)) -- We start the
reversing

```

Everything is $O(1)$ due to lazy evaluation. Recall that due to lazy evaluation, we don't complete `rotate` at one go, we do steps as and when required.

Again, this is an extremely neat idea and quite non-trivial. We recommend you internalize it and convince yourself of all the properties before moving ahead.

We will do one problem using queues before moving ahead. Also, another problem is provided for your practice and enjoyment.

[x Priest and Thieves](#)

[x A Lagged Fibonacci Sequence \(Project Euler 248\)](#)

A sequence is defined as:

$$g_k = 1 \quad \text{for } 0 \leq k \leq 1999 \quad g_k = g_{k-2000} + g_{k-1999} \quad \text{for } k \geq 2000$$

Find $g_k \bmod 20092010$ for $k = 10^{18}$.

§11.3. Binary Search Tree

A common problem we have is finding things, say integers, in a list. This is quite a task as `elem` is $O(n)$. Similarly, deleting something from a list is also quite costly as we need to find it, delete it and then rejoin the list.

Can we come up with a data structure which could allow us to do this faster? Consider the following data structure

```

data Btree t = Empty | Node t (Btree t) (Btree t)

```

This is called a binary tree. Now consider

≡ Binary Search Tree

A binary Search tree is a binary tree with the property:

- Every left descendant of a node has value less than that node.
- Every right descendant of a node has value larger than that node.

Thus, we can convert a list to a binary Search tree by simply doing almost a version of quick sort

```
lisToTree :: Ord a => [a] -> Btree a
lisToTree []      = Empty
lisToTree (x:xs) =
  Node x (lisToTree [l | l <- xs, l < x])
        (lisToTree [l | l <- xs, l > x])
```

We can also talk about the height of the tree aka the number of levels in the tree.

```
height :: BTree a -> Integer
height Empty = 0
height (Node a left right) = 1 + max (height left) (height right)
```

This clearly takes time $O(h)$ where h is equal to the height of the tree.

Moving ahead, we can insert and search the tree as follow:

```
insert :: Ord a => a -> Btree a -> Btree a
insert a Empty = Node a Empty Empty
insert a (Node x left right)
  | a == x    = Node x left right
  | a < x     = Node x (insert a left) right
  | otherwise = Node x left (insert a right)

search :: Ord a => a -> Btree a -> Bool
search a Empty = False
search a (Node x left right)
  | a == x    = True
  | a < x     = search a left
  | a > x     = search a right
```

Both these operations clearly take worst case $O(h)$ time where h is the height of the tree. Another, intrsting function is `findMin` and `findMax` which return the minimum and maximum element of the tree.

```
findMin :: Ord a => BTree a -> a
findMin Empty = error "empty tree"
findMin (Node a Empty Empty) = a
findMin (Node a Empty right) = a
findMin (Node a left _) = min a (findMin left)
```

x findMax

implement the function `findMax` which find's the maximum element of a binary search tree.

It is again easy to see that worst case time complexity is $O(h)$. We are now finally ready to implement the delete.

Deletion is somewhat trickier, because removing a node warrants appropriately linking its two descendant subtrees. We can be faced with three cases:

- If the doomed node is childless, we delete it and replace the void with `Empty`.
- If the doomed node has one child, we just move the child up the hierarchy.
- If the to-be-deleted node has two children, our solution is to relabel this node with the minimum element in its right descendants. This works as all elements in the right tree are greater than the deleted node and this is the lowest such node, maintaining the rightness of the right tree.

We implement this as:

```
delete :: Ord a => a -> Btree a -> Btree a
delete a Empty = Empty
delete a (Node x left right)
  | a == x = join left right
  | a > x = Node x left (delete a right)
  | a < x = Node x (delete a left) right

join :: Ord a => Btree a -> Btree a -> Btree a
join Empty Empty = Empty
join Empty r = r
join l Empty = l
join l r = let k = findMin r in Node k l (delete kr)
```

Note, we make 3 $O(h)$ operations. This makes delete also $O(h)$.

This all seems jolly good, the only issue is that based on the initial order insertions and deletions of the tree, the height could be of $O(n)$ and we end up doing no better.²⁴

The fact of the matter is, given n elements, we can make a binary search tree with height $\lceil \log_2 n \rceil = \lceil \log n \rceil$ which would get all the complexities to $O(\log n)$. This property is called balance.

So can we modify our insertion and deletion operators to keep the tree balanced without adding too much of additional complexity? As it turns out, yes.

The main idea is that we can rotate trees by choosing a different node and modifying the given tree to a binary tree. For example

```
Node x (Node y (Leaf a) (Leaf b)) (Leaf c)
  → Node y (Leaf a) (Leaf b) (Node x Empty Leaf c) (not binary, sad)
  → Node y (Leaf a) (Node x (Leaf b) Leaf c)
```

This is useful for say

```
Node 3 (Leaf 2) (Node 5 (Leaf 4) (Node 7 (Leaf 6) (Leaf 8)))
  → Node 5 (Node 3 (Leaf 2) (Leaf 4)) (Node 7 (Leaf 6) (Leaf 8))
```

An auxiliary idea is using a proxy for balance. Balance is a rather hard to maintain quality. Instead, we should choose some quantifiable way to say if a tree is balanced or not. It is especially nice if this proxy happens to be easy to compute and update.

In our case, we take the proxy for balance to be the difference in height of left and right trees. Our balance proxy will be that the difference in these heights is at most 1. Such trees are called height-balanced trees.

²⁴From the quick sort analysis, remember the average height of tree is $O(\log n)$, but the problem here is the worst case.

The rotation scheme we will use was created by Georgy Adelson-Velsky and Evgenii Landis in 1962 and the subsequent tree is called AVL trees. To aid some of the things we will do, we declare the data as:

```
data AvlTree = Empty | Node t Int (AvlTree t) (AvlTree t)
```

where the `Int` is there to store the height of the tree. We will now define some functions which will make our life easier:

```
height :: AvlTree t → Int
height Empty = 0
height (Node _ h _ _) = h

imbalance :: AvlTree t → Int
imbalance Empty = 0
imbalance (Node _ x tl tr) = height tl - height tr
```

We want `abs imbalance ≤ 1`.

We will now define two rotations, one which takes the left child and makes it the parent the other that takes the right child and makes it the parent.

```
rotateLeft :: AvlTree a → AvlTree a
rotateLeft (Node h x tl (Node hr y trl trr))
  = Node nh y (Node nhl x tl trl) trr where
    nhl = 1 + max (height tl) (height trl)
    nh  = 1 + max nhl (height trr)

rotateRight :: AvlTree a → AvlTree a
rotateRight (Node h x (Node hl y tll tlr) tr)
  = Node nh y tll (Node nhr x tlr tr) where
    nhr = 1 + max (height tlr) (height tr)
    nh  = 1 + max (height tll) nhr
```

We will now define a function `rebalance` which given a AVL tree with `abs imbalance == 2`, balance the tree. We only need this specific case as starting with a balanced tree, any operation can only increase the imbalance by 1 in either direction.

We will consider the following cases

- `imbalance == 2` and both subtrees are balanced : rotate the entire tree right
- `imbalance == 2` and imbalance of the left subtree is `-1` : left rotate the left subtree and then right rotate the tree
- `imbalance == -2` and both subtrees are balanced : rotate the entire tree left
- `imbalance == -2` and imbalance of right subtree is `1` : right rotate the right subtree and then left rotate the tree

This leads to

```

rebalance :: Ord a => AVLTree a -> AVLTree a
rebalance t@(Node h x tl tr)
  | abs st < 2 = t
  | st == 2 = if stl == -1 then
    rotateRight (Node h x (rotateLeft tl) tr)
  else rotateRight t
  | st == -2 = if str == 1 then
    rotateLeft (Node h x tl (rotateRight tr))
  else rotateLeft t
  where
    (st, stl, str) = (imbalance t, imbalance tl, imbalance tr)

```

Notice that rebalance is $O(1)$ as we are just moving things around via pattern matching.

```

insertAVL :: Ord a => a -> AVLTree a -> AVLTree a
insertAVL v Nil = Node 1 v Nil Nil
insertAVL v t@(Node h x tl tr)
  | v < x = rebalance (Node nhl x ntl tr)
  | v > x = rebalance (Node nhr x tl ntr)
  | v == x = t
  where
    ntl = insertAVL v tl
    ntr = insertAVL v tr
    nhl = 1 + max (height ntl) (height tr)
    nhr = 1 + max (height tl) (height ntr)

deleteMax :: Ord a => AVLTree a -> (a, AVLTree a)
deleteMax (Node _ x tl Nil) = (x, tl)
deleteMax (Node h x tl tr) = (y, rebalance (Node nh x tl ty))
  where
    (y, ty) = deleteMax tr
    nh = 1 + max (height tl) (height ty)

deleteAVL :: Ord a => a -> AVLTree a -> AVLTree a
deleteAVL v Nil = Nil
deleteAVL v t@(Node h x tl tr)
  | v < x = rebalance (Node nhl x ntl tr)
  | v > x = rebalance (Node nhr x tl ntr)
  | v == x = if isEmpty tl then tr
    else rebalance (Node nh y ty tr)
  where
    (y, ty) = deleteMax tl
    (ntl, ntr) = (deleteAVL v tl, deleteAVL v tr)
    nhl = 1 + max (height ntl) (height tr)
    nhr = 1 + max (height tl) (height ntr)
    nh y = 1 + max (height ty) (height tr)

```

We will not see any specific problems using the `BTree` or `AVLTree`, these are commonly used as precursors to more powerful data structures.

x Traversals and Sort

Write a function `traverse :: Ord a => AVLTree a -> [a]` which traverses the tree generating an increasing function.

Write a function `listToAVL :: Ord a => [a] -> AVLTree a` which converts a list to an AVL tree.

Write a function `treeSort :: Ord a => [a] -> [a]` which sorts a list using the avl tree. What is the time complexity of this?

§ 11.4. Sets and Maps

÷ Sets

A set is a data structure that can store unique values, without any particular order.

Notice that using a list as a set is suboptimal as insertion is $O(n)$ due to needing to check if the element is unique. But we can instead use an AVL tree to represent a set. This imposes the need for the elements to be `Ord`, circumventing it is beyond the scope of this book.

```
data Set a = Set (AVLTree a)
```

§ 11.5. Exercise

Type Classes

§12.1. Basics of Typeclasses

§12.1.1. Vector Spaces

We will assume throughout that our field of scalars is \mathbb{Q} .

And \mathbb{Q} is equivalent to `Rational`, which we make equivalent to `Q`.

```
type Q = Rational
```

Now, what makes something a vector space?

Well, `v` is a vector space if and only if

there is a function `vectorAddition :: v -> v -> v` which adds any 2 elements of `v`,

and another function `scalarMultiplication :: Rational -> v -> v` which takes a `Rational` number and “scales” an element of `v` by that number.

There is a way to express the above notion in Haskell!

A definition of `VectorSpace`

```
class VectorSpace v where
  vectorAddition :: v -> v -> v
  scalarMultiplication :: Q -> v -> v
```

This reads - “For a type `v` to be a `VectorSpace`, the functions `vectorAddition :: v -> v -> v` and `scalarMultiplication :: Rational -> v -> v` need to be defined.”

Now, we know that \mathbb{Q}^4 which is equivalent to `(Q,Q,Q,Q)` is a vector space. But why is that?

Well, it is so because we can define -

```
vectorAddition :: (Q,Q,Q,Q) -> (Q,Q,Q,Q) -> (Q,Q,Q,Q)
vectorAddition (p1,p2,p3,p4) (q1,q2,q3,q4) = (p1+q1,p2+q2,p3+q3,p4+q4)
```

and we can define

```
scalarMultiplication :: Q -> (Q,Q,Q,Q) -> (Q,Q,Q,Q)
scalarMultiplication q (q1,q2,q3,q4) = (q*q1,q*q2,q*q3,q*q4)
```

There is a particular way to communicate this to Haskell, which is as follows -

λ example instance of `VectorSpace`

```
instance VectorSpace (Q,Q,Q,Q) where

    vectorAddition :: (Q,Q,Q,Q) → (Q,Q,Q,Q) → (Q,Q,Q,Q)
    vectorAddition (p1,p2,p3,p4) (q1,q2,q3,q4) = (p1+q1,p2+q3,p3+q3,p4+q4)

    scalarMultiplication :: Q → (Q,Q,Q,Q) → (Q,Q,Q,Q)
    scalarMultiplication q (q1,q2,q3,q4) = (q*q1,q*q3,q*q3,q*q4)
```

This reads - “

`(Q,Q,Q,Q)` is a `VectorSpace`.

Why?

Because we give a definition for `vectorAddition :: (Q,Q,Q,Q) → (Q,Q,Q,Q) → (Q,Q,Q,Q)` and a definition for `scalarMultiplication :: Q → (Q,Q,Q,Q) → (Q,Q,Q,Q)` “

We know that `Q` itself is also a `VectorSpace`. So let's tell that to Haskell -

λ scalar field is `VectorSpace`

```
instance VectorSpace Q where

    vectorAddition :: Q → Q → Q
    vectorAddition = (+)

    scalarMultiplication :: Q → Q → Q
    scalarMultiplication = (*)
```

§12.1.2. Other Typeclasses

Haskell allows us to define other kinds of spaces as well, not just the notion of `VectorSpace`.

Just like

a type `v` is a `VectorSpace` if we can define `vectorAddition :: v → v → v` and `scalarMultiplication :: Q → v → v`,

a type `t` is said to be an `Eq` space if we can define a notion of equality `(==) :: t → t → Bool`.

Let's see the code for this -

λ definition of `Eq`

```
class Eq t where
    (==) :: t → t → Bool
```

This reads - “For a type `t` to be an `Eq` space, the function `(==) :: t → t → Bool` needs to be defined.”

Recall the type definition

```
data Colour = Red | Green | Blue
```

Now, we can make `Colour` an `Eq` space -

A Colour is an Eq space

```
instance Eq Colour where

    (==) :: Colour → Colour → Bool
    Red   == Red   = True
    Blue  == Blue  = True
    Green == Green = True
    _     == _     = False
```

This reads - “`Colour` is an `Eq` space, since we can define when two `Colour`s are equal or not by defining the function `(==) :: Colour → Colour → Bool`.”

Haskell also inbuilt definitions for -

```
(==) :: Integer → Integer → Bool
(==) :: Char → Char → Bool
(==) :: Bool → Bool → Bool
```

Thus `Integer`, `Char`, and `Bool` are `Eq` spaces.

Remember this datatype?

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

Any element of `Point` represents a point on the \mathbb{R}^2 plane.

We can tell Haskell that `Point` is also an `Eq` space -

```
instance Eq Point where

    (==) :: Point → Point → Bool
    point1 == point2 = x_coord point1 == x_coord point2
                    && y_coord point1 == y_coord point2
```

This reads - “`Point` is an `Eq` space, since we can define when two `Point`s are equal or not by defining that their `x_coord` inates need to be equal and their `y_coord` inates need to be equal.”

§12.1.3. General Definition

Such a notion of a “space” (like `VectorSpace` or `Eq` space), defined by the existence of certain functions (like `vectorAddition` or `(==)` respectively) is called a typeclass.

≡ typeclass

A typeclass is a *collection of types*,

and this collection should be the collection of all types `t` such that some polymorphic functions `f1`, `f2`, `f3`, ..., `fn` are defined for the type `t`.

≡ method

These functions `f1`, `f2`, `f3`, ..., `fn` are said to be **methods of the typeclass** that they help define.

So, to recapitulate, we have seen the **typeclass** `VectorSpace`, which is the collection of all types `v` for which the **methods** `vectorAddition` and `scalarMultiplication` is defined for `v`.

And we have also seen the **typeclass** `Eq`, which is the collection of all types `t` such that the **method** `(==)` is defined for `t`.

So, we now know how to define a **typeclass** in Haskell -

λ defining typeclass syntax

```
class MyTypeClass t where
  f1 :: <some type involving t>
  f2 :: <some type involving t>
  f3 :: <some type involving t>
  .
  .
  .
  fn :: <some type involving t>
```

And we also now know how to communicate to Haskell that some type `MyType` is included in this **typeclass** -

λ making a type an instance of typeclass

```
instance MyTypeClass MyType where

  f1 :: <type of f1 with t replaced by MyType>
  f1 = <some definition for f1>

  f2 :: <type of f2 with t replaced by MyType>
  f2 = <some definition for f2>

  f3 :: <type of f3 with t replaced by MyType>
  f3 = <some definition for f3>

  .
  .
  .

  fn :: <type of fn with t replaced by MyType>
  fn = <some definition for fn>
```

Basically, by definition,

`MyType` is included in `MyTypeClass`

only when the **methods** of `MyTypeClass` are defined for `MyType`,
so we just have to provide the appropriate definitions for those **methods**.

§12.2. The Point of Typeclasses

§12.2.1. The Point of Vector Spaces

Why do we define the concept of a vector space? Well, historically, it is because people noticed that we can prove things for vector spaces in general.

For example -

We can prove that \mathbb{Q}^4 has a vector space basis.

We can prove that \mathbb{Q}^5 has a vector space basis.

We can prove that \mathbb{Q}^6 has a vector space basis.

We can prove that \mathbb{Q}^7 has a vector space basis.

.

.

.

In fact, we can prove that $\mathbb{Q}^{\mathbb{N}}$ (the set of all functions from \mathbb{N} to \mathbb{Q}) has a vector space basis.

But it is “easier” to just prove in general that all vector spaces have a vector space basis.

This general proof will apply to all vector spaces. So the concept of a vector space is useful because it allows us to talk about and prove properties of many objects at once.

The above is one of the reasons why we define vector spaces.

§12.2.2. The Point of any Class

Similarly, \div **typeclasses** in Haskell allow us to write function definitions for many types at once.

Let’s suppose we wanted to define vector subtraction. What would that look like?

```
λ vectorSubtraction
vectorSubtraction v1 v2 =
  v1 `vectorAddition` ( (-1) `scalarMultiplication` v2 )
-- basically
-- v1 - v2 == v1 + ( -v2 ) == v1 + ( (-1) * v2 )
```

This definition should apply in any **VectorSpace**, right?

Therefore we’ve written a function definition that works for any type which is a **VectorSpace**.

Recall the definition of the λ **elem** function, used to answer whether a given object appears in a given list -

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

So long as **x** and **y** are from an **Eq** space, we can do **x == y** and this function will be defined.

Therefore we’ve written a function definition that works for any list over any **Eq** space.

≡ ad-hoc polymorphism

In summary, as long as we only fundamentally refer to the ≡ methods of a ≡ typeclass, we can define functions that work for any type in that ≡ typeclass.

This is called **ad-hoc polymorphism**

§12.2.2.1. Types

As discussed, the function λ `vectorSubtraction` should work for any type which is a `VectorSpace`. But then what is the type of the function?

It should be `vectorSubtraction :: v → v → v`, where `v` is any `VectorSpace`.

Haskell's way of saying the above is as follows -

```
>>> :type vectorSubtraction
vectorSubtraction :: VectorSpace v => v -> v -> v
```

It adds to `VectorSpace v =>` to say that `v` can be any `VectorSpace`.

Similarly for λ `elem` -

```
>>> :type elem
elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

For now, `Foldable` just means something which is “like a list”.

§12.3. Induced Instances

We know that if V and W are vector spaces, then the set $V \times W$ can be given a vector space structure as follows -

$$\begin{aligned}(v_1, w_1) + (v_2, w_2) &:= (v_1 + v_2, w_1 + w_2) \\ c(v, w) &:= (cv, cw)\end{aligned}$$

This is known as the external direct sum of vector spaces $V \oplus W$.

We can express this notion in Haskell as follows -

```
instance (VectorSpace v, VectorSpace w) => VectorSpace (v,w) where

  vectorAddition :: (v,w) -> (v,w) -> (v,w)

  (v1,w1) `vectorAddition` (v2,w2) =
    ( v1 `vectorAddition` v2 , w1 `vectorAddition` w2 )

  scalarMultiplication :: Q -> (v,w) -> (v,w)

  c `scalarMultiplication` (v,w) =
    ( c `scalarMultiplication` v , c `scalarMultiplication` w )
```

This reads - “If v and w are `VectorSpace`s, then (v,w) is a `VectorSpace` because we define `vectorAddition` and `scalarMultiplication` for (v,w) like so...”

Similarly, if t and s are `Eq` spaces, then (t,s) is also an `Eq` space -

```
instance (Eq t, Eq s) => Eq (t,s) where
  (t1,s1) == (t2,s2) = ( t1 == t2 ) && ( s1 == s2 )
```

This reads - “If t and s are `Eq` spaces, then (t,s) is a `Eq` space because we define `vectorAddition` and `scalarMultiplication` for (v,w) like so...”

If t is an `Eq` space, then $[t]$ is also an `Eq` space -

```
instance Eq t => Eq [t] where
  [] == [] = True
  (t:ts) == (t':ts') = ( t == t' ) && ( ts == ts' )
  _ == _ = False
```

And so on...

§12.4. The `deriving` Keyword

In class and tutorial

§12.5. Superclasses and Subclasses

There is a `typeclass` denoted `Ord`, which means “`Ord`ered space”.

A type t is an `Ord`ered space (i.e., t is in the `Ord` `typeclass`) if and only if `==` and `<` are defined for t .

Thus we can recognize that `==` and `<` are the `methods` for the `Ord` `typeclass`.

That means every `Ord`ered space has `==` defined for it. Which means every `Ord`ered space is an `Eq` space.

Such a `typeclass` is called a “subclass”.

`subclass`

If the set of `methods` of a typeclass is a proper superset of the set of `methods` of another typeclass, then the former typeclass is said to be a `subclass` of the latter.

The syntax for defining a `subclass` is a little different from usual. For example, let’s define `Ord` -

```
instance Eq o => Ord o where
  (<) :: o -> o -> Bool
```

Notice that we didn’t mention `==` in the above definition, even though it’s a `method` of `Ord`.

This is because it is already defined in `Eq`, and we make sure that o should also be in an `Eq` space by the `Eq o =>`.

§12.6. Laws

Is `Bool` a `VectorSpace`? No, of course not!

But what about this -

```
instance VectorSpace Bool where

    vectorAddition :: Bool → Bool → Bool
    vectorAddition = (/=)

    scalarMultiplication :: ℚ → Bool → Bool
    scalarMultiplication c bool = ( c /= 0 ) && bool
```

Haskell now believes that `Bool` is a `VectorSpace`.

So what gives?

Well, these addition and scaling functions don't satisfy the vector space axioms.

The vector space axioms are -

$$\begin{aligned} &\forall \text{ vectors } u, v, w \text{ and scalars } c, d \\ &(v + w) + u == v + (w + u) \\ &c \cdot (d \cdot v) == (c \cdot d) \cdot v \\ &c \cdot (v + w) == c \cdot v + c \cdot w \\ &(c + d) \cdot v == c \cdot v + d \cdot v \\ &\vdots \end{aligned}$$

etc.

But we have that

$$(1 + 1) \cdot \text{True} \neq 1 \cdot \text{True} + 1 \cdot \text{True}$$

which violates the axiom

$$(c + d) \cdot v == c \cdot v + d \cdot v$$

We should ensure - that for any supposed `VectorSpace`, the \neq methods we define satisfy the axioms, otherwise we might end up with nonsense like `Bool` being a `VectorSpace`.

However, these kinds of problems can occur not just for `VectorSpace`, but for other \neq typeclasses as well -

For example, consider this following nonsense giving a wrong definition of `Bool` being an `Eq` space -

```
instance Eq Bool
    (==) :: Bool → Bool → Bool
    False == False = True
    False == True  = False
    True  == False = True
    True  == True  = True
```

Here, we have defined `True == False` but `False /= True`.

This violates the “axiom” of “symmetricity”, which is - `if x==y then y==x`

≡ **typeclass laws**

Thus we should ensure - that for any type supposed to be an instance of a ≡ **typeclass**, the
≡ **methods** we define satisfy some “*axioms*”, otherwise we might end up with nonsense.

These “*axioms*” are called **Laws**.

Monads

Appendix

Nothing past this point is for exam, obviously.

§ 14.1. Preface

- [return to the above heading](#)

§ 14.2. Why you should care about Haskell

- [return to the above heading](#)

§ 14.2.1. If You Like Programming

- [return to the above heading](#)

§ 14.2.1.1. Influence on Programming Language Design

- [return to the above heading](#)

§ 14.2.1.2. Understandability, Readability and Refactorability

- [return to the above heading](#)

§ 14.2.1.3. Makes you a Better Programmer

- [return to the above heading](#)

§ 14.2.1.4. Catches Bugs before Running

- [return to the above heading](#)

§ 14.2.1.5. Builds Safety-Critical Software

- [return to the above heading](#)

§ 14.2.1.6. Used by Prestigious People and Organizations

- [return to the above heading](#)

§ 14.2.2. If You Like Mathematics

- [return to the above heading](#)

§ 14.2.2.1. Haskell is Math

- [return to the above heading](#)

§ 14.2.2.2. Math begets Haskell

- [return to the above heading](#)

§ 14.2.2.3. Haskell begets Math

- [return to the above heading](#)

§ 14.2.2.4. Proof Assistants

- [return to the above heading](#)

§ 14.2.2.5. Used in Mathematical Computation

- [return to the above heading](#)

§ 14.2.2.6. Expressivity of Haskell

- [return to the above heading](#)

§ 14.3. How to Learn Haskell

- [return to the above heading](#)

§ 14.3.1. If You Like Math

- [return to the above heading](#)

§ 14.3.2. If You Like Programming

- [return to the above heading](#)

§ 14.4. How to Read this Book

- [return to the above heading](#)

§ 14.5. Table of Contents

- [return to the above heading](#)

§ 14.6. Basic Theory

- [return to the above heading](#)

§ 14.7. Precise Communication

- [return to the above heading](#)

§ 14.8. The Building Blocks

- [return to the above heading](#)

§ 14.9. Values

- [return to the above heading](#)

§ 14.10. Variables

- [return to the above heading](#)

§ 14.11. Well-Formed Expressions

- [return to the above heading](#)

§ 14.12. Function Definitions

- [return to the above heading](#)

§ 14.12.1. Using Expressions

- [return to the above heading](#)

§ 14.12.2. Some Conveniences

- [return to the above heading](#)

§ 14.12.2.1. Where, Let

- [return to the above heading](#)

§ 14.12.2.2. Anonymous Functions

- [return to the above heading](#)

§ 14.12.2.3. Piecewise Functions

- [return to the above heading](#)

§ 14.12.2.4. Pattern Matching

- [return to the above heading](#)

§ 14.12.3. Recursion

- [return to the above heading](#)

§ 14.12.3.1. Termination

- [return to the above heading](#)

§ 14.12.3.2. Induction

- [return to the above heading](#)

§ 14.12.3.3. Proving Termination using Induction

- [return to the above heading](#)

§ 14.13. Infix Binary Operators

- [return to the above heading](#)

§ 14.14. Trees

- [return to the above heading](#)

§ 14.14.1. Examples of Trees

- [return to the above heading](#)

§ 14.14.2. Making Larger Trees from Smaller Trees

- [return to the above heading](#)

§ 14.14.3. Formal Definition of Trees

- [return to the above heading](#)

§ 14.14.4. Structural Induction

- [return to the above heading](#)

§ 14.14.5. Structural Recursion

- [return to the above heading](#)

§ 14.14.6. Termination

- [return to the above heading](#)

§ 14.15. Why Trees?

- [return to the above heading](#)

§ 14.15.1. The Problem

- [return to the above heading](#)

§ 14.15.2. The Solution

- [return to the above heading](#)

§ 14.15.3. Exercises

- [return to the above heading](#)

§ 14.16. Installing Haskell

- [return to the above heading](#)

§ 14.17. Installation

- [return to the above heading](#)

§ 14.17.1. General Instructions

- [return to the above heading](#)

§ 14.17.2. Choose your Operating System

- [return to the above heading](#)

§ 14.17.2.1. Linux

- [return to the above heading](#)

§ 14.17.2.2. MacOS

- [return to the above heading](#)

§ 14.17.2.3. Windows

- [return to the above heading](#)

§ 14.18. Running Haskell

- [return to the above heading](#)

§ 14.19. Fixing Errors

- [return to the above heading](#)

§ 14.20. Autocomplete

- [return to the above heading](#)

§ 14.21. Basic Syntax

- [return to the above heading](#)

§ 14.22. The Building Blocks

- [return to the above heading](#)

§ 14.23. Values

- [return to the above heading](#)

§ 14.24. Variables

- [return to the above heading](#)

§ 14.25. Types

- [return to the above heading](#)

§ 14.25.1. Using GHCi to get Types

- [return to the above heading](#)

§ 14.25.2. Types of Functions

- [return to the above heading](#)

§ 14.26. Well-Formed Expressions

- [return to the above heading](#)

§ 14.27. Infix Binary Operators

- [return to the above heading](#)

§ 14.27.1. Precedence

- [return to the above heading](#)

§ 14.28. Logic

- [return to the above heading](#)

§ 14.28.1. Truth

- [return to the above heading](#)

§ 14.28.2. Statements

- [return to the above heading](#)

§ 14.29. Conditions

- [return to the above heading](#)

§ 14.29.1. Logical Operators

- [return to the above heading](#)

§ 14.29.1.1. Exclusive OR aka XOR

- [return to the above heading](#)

§ 14.30. Function Definitions

- [return to the above heading](#)

§ 14.30.1. Using Expressions

- [return to the above heading](#)

§ 14.30.2. Some Conveniences

- [return to the above heading](#)

§ 14.30.2.1. Piecewise Functions

- [return to the above heading](#)

§ 14.30.2.2. Pattern Matching

- [return to the above heading](#)

§ 14.30.2.3. Where, Let

- [return to the above heading](#)

§ 14.30.2.4. Without Inputs

- [return to the above heading](#)

§ 14.30.2.5. Anonymous Functions

- [return to the above heading](#)

§ 14.30.3. Recursion

- [return to the above heading](#)

§ 14.31. Optimization

- [return to the above heading](#)

§ 14.32. Numerical Functions

- [return to the above heading](#)

§ 14.32.1. Division, A Trilogy

- [return to the above heading](#)

§ 14.32.2. Exponentiation

- [return to the above heading](#)

§ 14.32.3. `gcd` and `lcm`

- return to the above heading

§ 14.32.4. Dealing with Characters

- return to the above heading

§ 14.33. Mathematical Functions

- return to the above heading

§ 14.33.1. Binary Search

- return to the above heading

§ 14.33.2. Taylor Series

- return to the above heading

§ 14.34. Exercises

- return to the above heading

§ 14.35. Types as Sets

- return to the above heading

§ 14.36. Sets

- return to the above heading

§ 14.37. Types

- return to the above heading

§ 14.37.1. $::$ is analogous to \in or \ni belongs

- return to the above heading

§ 14.37.2. $A \rightarrow B$ is analogous to B^A or \ni set exponent

- return to the above heading

§ 14.37.3. (A, B) is analogous to $A \times B$ or \ni cartesian product

- return to the above heading

§ 14.37.4. $()$ is analogous to \ni singleton set

- return to the above heading

§ 14.37.5. No \ni intersection of Types

- return to the above heading

§ 14.37.6. No \ni union of Types

- return to the above heading

§14.37.7. Disjoint Union of Sets

- return to the above heading

§14.37.8. `Either A B` is analogous to $A \sqcup B$ or \div disjoint union

- return to the above heading

§14.37.9. The `Maybe` Type

- return to the above heading

§14.37.10. `Void` is analogous to $\{\}$ or \div empty set

- return to the above heading

§14.38. Currying

- return to the above heading

§14.38.1. In Haskell

- return to the above heading

§14.38.2. Understanding through Associativity

- return to the above heading

§14.38.2.1. Of \rightarrow

- return to the above heading

§14.38.2.2. Of Function Application

- return to the above heading

§14.38.2.3. Operator Currying Rule

- return to the above heading

§14.38.3. Proof of the Currying Theorem

- return to the above heading

§14.39. Exercises

- return to the above heading

§14.40. Introduction to Lists

- return to the above heading

§14.41. Type of List

- return to the above heading

§14.42. Creating Lists

- return to the above heading

§ 14.42.1. Empty List

- return to the above heading

§ 14.42.2. Arithmetic Progression

- return to the above heading

§ 14.43. Functions on Lists

- return to the above heading

§ 14.44. List Comprehension

- return to the above heading

§ 14.44.1. Cons or `(:)`

- return to the above heading

§ 14.45. Length

- return to the above heading

§ 14.45.1. Concatenate or `(++)`

- return to the above heading

§ 14.45.2. Head and Tail

- return to the above heading

§ 14.45.3. Take and Drop

- return to the above heading

§ 14.45.4. `(!!)`

- return to the above heading

§ 14.45.5. List `→ Bool`

- return to the above heading

§ 14.45.5.1. Elem

- return to the above heading

§ 14.45.5.2. Generalized Logical Operators

- return to the above heading

§ 14.46. Strings

- return to the above heading

§ 14.47. Structural Induction for Lists

- return to the above heading

§ 14.48. Sorting

- [return to the above heading](#)

§ 14.49. Optimization

- [return to the above heading](#)

§ 14.50. Lists as Syntax Trees

- [return to the above heading](#)

§ 14.51. Dark Magic

- [return to the above heading](#)

§ 14.51.1. Exercises

- [return to the above heading](#)

§ 14.52. Polymorphism and Higher Order Functions

- [return to the above heading](#)

§ 14.53. Polymorphism

- [return to the above heading](#)

§ 14.53.1. Classification has always been about *shape* and *behaviour* anyway

- [return to the above heading](#)

§ 14.53.2. A Taste of Type Classes

- [return to the above heading](#)

§ 14.54. Higher Order Functions

- [return to the above heading](#)

§ 14.54.1. Currying

- [return to the above heading](#)

§ 14.54.2. Functions on Functions

- [return to the above heading](#)

§ 14.54.3. A Short Note on Type Inference

- [return to the above heading](#)

§ 14.54.4. Higher Order Functions on Maybe Type : A Case Study

- [return to the above heading](#)

§ 14.55. Advanced List Operations

- [return to the above heading](#)

§14.56. List Comprehensions

- [return to the above heading](#)

§14.57. Zip it up!

- [return to the above heading](#)

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

- [return to the above heading](#)

§14.58.1. Orgami of Code!

- [return to the above heading](#)

§14.58.2. Numerical Integration

- [return to the above heading](#)

§14.58.3. Time to Scan

- [return to the above heading](#)

§14.59. Excercises

- [return to the above heading](#)

§14.60. Introduction to Datatypes

- [return to the above heading](#)

§14.61. Datatypes (Once Again)

- [return to the above heading](#)

§14.62. Finite Types

- [return to the above heading](#)

§14.63. Product Types

- [return to the above heading](#)

§14.64. Parametric Types

- [return to the above heading](#)

§14.65. Sum Types

- [return to the above heading](#)

§14.66. Inductive Types

- [return to the above heading](#)

§14.66.1. Inductive Types (as a Mathematician)

- [return to the above heading](#)

§ 14.66.1.1. Natural Numbers as Inductive Types

- [return to the above heading](#)

§ 14.66.1.2. Lists as Inductive Types

- [return to the above heading](#)

§ 14.66.2. (Not Quite) Inductive Types (as a Programmer)

- [return to the above heading](#)

§ 14.66.2.1. Calculator

- [return to the above heading](#)

§ 14.66.2.2. Trees as Inductive Types

- [return to the above heading](#)

§ 14.66.2.3. Binary Trees

- [return to the above heading](#)

§ 14.66.2.4. Addressing the “Not Quite” ...

- [return to the above heading](#)

§ 14.67. Same Same but Different

- [return to the above heading](#)

§ 14.67.1. Same Same

- [return to the above heading](#)

§ 14.67.2. Different

- [return to the above heading](#)

§ 14.68. Computation as Reduction

- [return to the above heading](#)

§ 14.69. Complexity

- [return to the above heading](#)

§ 14.70. Introduction

- [return to the above heading](#)

§ 14.71. Asymptotics

- [return to the above heading](#)

§ 14.71.1. Big El

- [return to the above heading](#)

§ 14.71.2. The Hierarchy of Functions

- [return to the above heading](#)

§ 14.71.3. Big Oh notation

- [return to the above heading](#)

§ 14.72. Asymptotic Mathematics

- [return to the above heading](#)

§ 14.73. Analysis of Algorithms

- [return to the above heading](#)

§ 14.73.1. Sorting

- [return to the above heading](#)

§ 14.73.1.1. Counting Sort

- [return to the above heading](#)

§ 14.73.1.2. Radix Sort

- [return to the above heading](#)

§ 14.73.1.3. Survey of Sorting Algorithms

- [return to the above heading](#)

§ 14.74. RAM Model and Asymptotic Analysis

- [return to the above heading](#)

§ 14.74.1. What Big O doesn't want you to know?

- [return to the above heading](#)

§ 14.75. An Informal Survey of Multiplication Algorithms

- [return to the above heading](#)

§ 14.76. Advanced Data Structures

- [return to the above heading](#)

§ 14.77. Datatypes (One last time)

- [return to the above heading](#)

§ 14.78. Stacks and Queues

- [return to the above heading](#)

§ 14.78.1. Stack

- [return to the above heading](#)

§ 14.78.2. Queue

- [return to the above heading](#)

§ 14.78.2.1. Amortization

- [return to the above heading](#)

§ 14.78.2.2. The True Queue

- [return to the above heading](#)

§ 14.79. Binary Search Tree

- [return to the above heading](#)

§ 14.80. Sets and Maps

- [return to the above heading](#)

§ 14.81. Exercise

- [return to the above heading](#)

§ 14.82. Type Classes

- [return to the above heading](#)

§ 14.83. Basics of Typeclasses

- [return to the above heading](#)

§ 14.83.1. Vector Spaces

- [return to the above heading](#)

§ 14.83.2. Other Typeclasses

- [return to the above heading](#)

§ 14.83.3. General Definition

- [return to the above heading](#)

§ 14.84. The Point of Typeclasses

- [return to the above heading](#)

§ 14.84.1. The Point of Vector Spaces

- [return to the above heading](#)

§ 14.84.2. The Point of any Class

- [return to the above heading](#)

§ 14.84.2.1. Types

- [return to the above heading](#)

§ 14.85. Induced Instances

- [return to the above heading](#)

§ 14.86. The **deriving** Keyword

- [return to the above heading](#)

§ 14.87. Superclasses and Subclasses

- [return to the above heading](#)

§ 14.88. Laws

- [return to the above heading](#)

§ 14.89. Monads

- [return to the above heading](#)

§ 14.90. Appendix

- [return to the above heading](#)