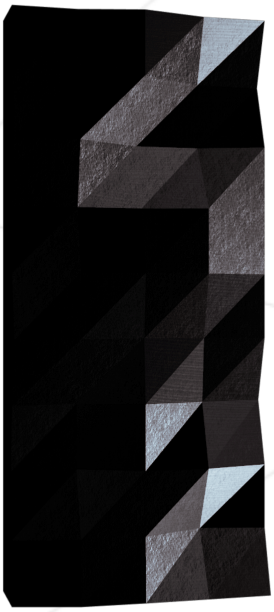


Ready, Set, Boole!

An introduction to Boolean Algebra

Tristan Duquesne, Luc Lenôtre

Summary: Discover the way computers work mathematically

Version: 2

Contents

Chapter I

Foreword

Even if we know today that many animals have an innate understanding of geometry and numbers, and that almost all civilizations have left traces of counting or geometry, these are not really "mathematics" as we understand it today. The history of mathematics, and with it, of science, "truly" begins, so to speak, in the Fertile Crescent. However, it was the work of several Greek thinkers that reoriented mathematics from these "primitive computational tools" to the powerful formal systems that gave rise to modern science.

The beginning of this transition took place through two thinkers in particular, Euclid and Aristotle.

Euclid could not bear to defend an idea without being able to prove it. In the end, the best Euclid could do was to limit the presuppositions (also: "postulates" or "axioms", things that are assumed to be true) of his theoretical monument to just 5 ideas: he still managed to build all the rest of his geometry depending **only** on these 5 ideas. His life's work, **The Elements**, is the second most published and translated book in human history (the first being the Bible).

Aristotle, both philosopher and scientist (although this distinction did not exist in his time) had an extremely prolific intellectual life, going much further than "pure" mathematics. Among other things, in his work called the **Organon**, Aristotle defines a method to correctly compute "truth": the **Organon** is the first treatise on logic as we understand it today. In it, he considered, among other things, that a logical system cannot have a contradiction, otherwise any idea within the system becomes both true and false (the "principle of explosion").

The echoes of these works travelled through the ages; Al-Khwarizmi, St. Thomas Aquinas... Contributions from other civilizations, especially from India, and the work of a number of mathematicians that is difficult to count, all contributed to strengthen the language of these formal systems (algebra) in order to make them the foundation of all modern sciences. However, the most fundamental ideas were developed very early. The combination of these two doctrines (Euclid's rigor, and Aristotle's logic) sowed the seeds that allowed mathematics to become the fundamental tool of human knowledge.

At the end of the 19th century, with the 2nd industrial revolution, a great sense of confidence took hold of Western scientists, who were then at the forefront of world in scientific research. The power of science and technology affirmed itself, and scientists started to believe that everything, absolutely everything, will be demonstrable by science: this ideology was dubbed positivism. Riemann, Lobachevsky, etc., succeeded in solving Euclid's main concern, namely to show that his 5th axiom had alternatives. And following the paths traced by George Boole, who figured out how to algebraize Aristotle's logic, and Georg Cantor, who with his theory of sets gave the world a common language to all mathematics; the faculties of philosophy and mathematics of the West allied themselves to give birth to the modern field of (computational) logic.

In 1900, during the International Congress of Mathematicians at the Sorbonne, David Hilbert (one of the most prolific and visionary mathematicians in history; the superstar of the time) gave a conference which has become the stuff of legends. In it, he declared 23 mathematical problems, which according to him, would define the XXth century (and he was largely right); affirming that the unknown would give way to mathematical science. Some of them are still unsolved today, and are already also legendary (some even have a pot of gold at the end of the rainbow...). But one of the most important, the 2nd problem, concerned the foundations of mathematics itself. It was a question of showing that the axioms of modern mathematics (set theory, along with the axioms of Zermelo and Fraenkel), which allowed one to construct arithmetic starting from set theory, led to a well-founded, consistent theory: one without any contradiction.

In 1936, an Austrian by the name of Kurt Gödel, a rising star in this community of logicians, answered Hilbert's 2nd problem in the negative: one cannot demonstrate the consistency of arithmetic. He starts from the fundamental axiomatic minimum: roughly speaking, "let there be a theory T expressed in a formal system (logical language) L, etc."; and shows that if this theory is rich enough to express arithmetic, then it is impossible to demonstrate the consistency of the theory with only the tools of the theory itself.

His proof, which is today referred to as the "incompleteness theorems", has an extremely technical formulation, but we will try to give you an idea. For that, we need some vocabulary.

A "proposition" is a sentence that can be true or false: "the weather is pleasant" is a proposition; "who am I?" is not. A proposition is said to be "provable" or "demonstrable" if there is a formal proof (a logical calculation) that this proposition is true. A proposition is said to be "refutable" if there is a formal proof that this proposition is false. A proposition that is either demonstrable or refutable is said to be "decidable". A proposition that is neither demonstrable nor refutable is said to be "undecidable".

Theorem 1: "any system (L,T) rich enough to formulate and perform some arithmetic is incomplete". In other words, there exists, in any language L with a theory T, defined according to certain axioms to be able to express "1+1=2", at least one proposition that is undecidable. For this, Gödel uses the proposition G: "the theorem G is not provable in the theory T", inspired by the liar's paradox ("this sentence is false": if it is true, then it is false; if it is false, it is true: contradiction). If G is provable, then G is true, so G is not provable: contradiction. If G is refutable, then G is false, so it is "not-not provable", so it is provable: contradiction. Conclusion, G is neither provable nor refutable: G is undecidable.

Theorem 2: If the system (L, T) contains arithmetic, and is consistent (contains no contradiction; Gödel "assumes that it can be done", ie, that if there exists a proof of consistency, it is a demonstration and not a refutation), then it is impossible to obtain a proof of the consistency of (L, T) by using only (L, T). Why not? Because if a proof of the coherence of (L,T) existed, it would allow, for any proposition of (L,T), to distinguish if this proposition is true, or if it is false. In other words, if such a proof existed, we could know, by passing through this proof, if G is true or false. But G is undecidable; so there is no such proof. Thus, there is no proof of the consistency of (L,T) within (L,T). In fact, if such a proof exists, there is a contradiction, and the principle of explosion applies.

The incompleteness theorems were an extreme shock for the mathematical community, and for science in general. They killed scientific positivism in its heart: even in mathematics, it is not possible to know everything. It was more than 2500 years of scientific, epistemological, and religious questioning that found their answer at this moment: one must make a choice of axioms, to base oneself on a "belief", even in math...

But some people, far from being defeated, reformulated the question. "We can't know everything", sure. But then, thanks to logic, we realized that "proving" and "calculating" were in fact two sides of the same coin; two ways of seeing the same universal phenomenon. As a result, the question "what can I know?" morphed into "what can I calculate?". And so, over the ashes of Hilbert's program and positivism, taking up the work of Polish, French, German, English and American logicians and engineers, Alan Turing and John von Neumann defined the notion of computation, and gave birth to modern computing, closing one era of human knowledge and giving birth to a new one.

Additional resources:

- **LogiComix** (Apostolos Doxiadis et alii): a phenomenal comic book dealing precisely with the period from the emergence of academic logic to the birth of computer science.
- **Gödel, Escher, Bach, an Eternal Golden Braid** (Douglas Hofstadter): a monument of literature, using the incompleteness theorems as an engine for analyzing self-referential systems, especially human cognition.
- **A History of Non-Euclidean Geometry** (Extra Credits): a series of 5 videos, less than an hour in total, that trace the history of mathematics using the historical continuations of Euclid's work as a guideline.
- **Math has a fatal flaw** (Veritasium): a great video that will walk you through a more technical understanding of Gödel's incompleteness theorems, and how Gödel uses the links between arithmetic and logic to prove his arguments.
- **How to count past infinity** (VSauce): a video which will explain how, through set theory, mathematicians realized there were distinct types of infinity.

Funny story: towards the end of his life, Einstein would often say that the reason he resided at the IAS in Princeton was only to be able to accompany Gödel on his walks and talk about mathematics with him.

I.1 How, through logic, we have tricked rocks into thinking

Post from user **gnhicbfjnjjjbb** on Reddit:

> Think of a simple light switch. Turns a lightbulb on or off. Now instead of hooking that switch up to a lightbulb, hook it up to another light switch.
>
> Suppose, for example, turning on the light switch makes the other light switch always send an "on" signal, even if it's off. That's a bit like what we'd call an "or gate", because only the first switch OR the second switch needs to be on in order to send an on signal from the second switch.
>
> We can also have the concept of "and", if we imagine that turning off the first switch completely cuts the circuit for the second switch, so even if the second switch is on, it doesn't turn the light on.
>
> Once we have "and" and "or" (well, also "not", but "not" is just an upside-down switch that turns things on if it's off, and off if it's on), we can calculate anything we want. For example, here's how we'd do simple arithmetic:
>
> (this is going to get a bit dense, but stick with me, because it's really important that computers are able to do this)
>
> First, convert the number into a "binary representation". This is a fancy way of saying "give each number a label that's a pattern of 'on's and 'off's." For example, we can represent each number 0 to 3 as 00, 01, 10, 11. In our world, we go 1 2 3 4 5 6 7 8 9 10, but in binary we pretend those middle digits don't exist, and instead of writing 2 as "2", we write it as "10". It still means 2 though - and now it's easier to represent with "on"s and "off"s.
>
> Second, we want to add just like normal adding. Let's look at just the rightmost digit - it can be either 0 or 1, and we're trying to add it to another digit that's either 0 or 1. At first, we might try something like an "or gate". Then 0+0 is 0, 0+1 is 1, and 1+0 is 1, which looks good so far. Except that 1 OR 1 will... also give us 1, which we don't want. We want to get 0 and carry a 1 to the left (remember, we can't create the digit 2, we have to represent 2 as "10"). So what we actually want is something called a "xor", that's a fancy name for "'or' and not 'and'". We take the result of an "or" gate, and we "and" it with the flipped result of an "and" gate. So we will have 0+0 = 0, 0+1 = 1, 1+0 = 1, and 1+1 = 0. To make sure we're actually adding 1 and 1, and not just erasing it to 0, we also need to record a carry digit, but that's just an "and" gate. If both the first AND second thing are 1, carry a 1, otherwise carry a 0.
>
> Third, we do the same thing one step to the left, but we also include the carry digit if we have one. We "xor" the digits to see if we should record

a 1 or a 0 in this position, and if we have 2 or more 1s (in gates we know, one way to write that is "'a and b' or 'b and c' or 'a and c'") we carry a 1 to the next position.

So we can do addition. With repeated addition, we can multiply. We can also do subtraction by a similar process. With repeated subtraction, we can do long division. So basically we can solve any math problem we want.

But how does a regular human trigger that math, if all the numbers are these weird sequences of "on" and "off"? Well, we can hook a few of the light switches back up to lightbulbs, but make them super tiny lightbulbs of different colors. That's screen pixels. If the light switches want to show the binary number "11", they can light up a pattern on the screen which looks like "3", so the human can understand it. How does the computer know what a "3" looks like? Well, the on-off patterns of that look like a "3" are represented as a big math formula, and our computer can do any math it wants to, so it can compute which lightbulbs (pixels) it needs to eventually turn on and off.

Under the hood, every piece of data - every image, every word - is represented with a numeric label of some kind, and it goes through a looooooong chain of on/off switches to turn it into an intelligible pattern of pixels on the screen.

A lot of it is a bunch of really really fast arithmetic. For example, if you can compute the paths of rays of light, you can draw a 3D picture on the screen. You do a bunch of physics equations about how the light would bounce off the object and into people's eyes, if they were looking at a real 3D object. But we know how our computer does math - it's a bunch of on-off switches hooked up together.

All those on-off switches are bits of wire on a piece of silicon, so that's how we tricked rocks into thinking.

Chapter II

Introduction

In this module we will talk about Boolean Algebra and Set Theory, both of which are very important theories in both mathematics and computer science.

Also, you will see that you probably already have a good grasp of Boolean Algebra without even knowing about it! This will be the occasion to push your understanding to the next level.

II.1 General Rules

- For this module, function prototypes are described in Rust, but you may use the language you want. There are a few constraints for your language of choice, though:
  - It must support generic types
  - It must support functions as first class citizens (example: supports lambda expressions)
  - It must natively implement bitwise operations over integer types, or at least bitwise operations over a bitmap/boolmap type; though the former is strongly recommended
  - Optional: support for operator overloading
- We recommend that you use paper if you need to. Drawing visual models, making notes of technical vocabulary... This way, it will be easier to wrap your head around the new concepts you will discover, and use them to build your understanding for more and more related concepts. Also, doing computations by head is really hard and error-prone, so having all the steps written down helps you figure out where you made a mistake when you did.
- Don't stay stuck because you don't know how to solve an exercise: make use of peer-learning! You can ask for help on Slack (42 or 42-AI) or Discord (42-AI) for example.

- You are not allowed to use any mathematical library, even the one included in your language's standard library, unless explicitly stated otherwise.

- For every exercise you may have to respect a given time and/or space complexity. These will be checked during peer review.

    - The time complexity is calculated relative to the number of executed instructions

    - The space complexity is calculated relative to the maximum amount of memory allocated simultaneously

    - Both have to be calculated against the size of the function's input (a number, the length of a string, etc...)

Chapter III

Exercise 00 - Adder

III.0.1 Goal

You must write a function that takes as parameters two natural numbers `a` and `b` and returns one natural number that equals `a + b`. However the only operations you're allowed to use are:

- `&` (bitwise AND)
- `|` (bitwise OR)
- `^` (bitwise XOR)
- `<<` (left shift)
- `>>` (right shift)
- `=` (assignment)
- `==`, `!=`, `<`, `>`, `<=`, `>=` (comparison operators)

The incrementation operator (`++` or `+= 1`) is allowed **only** to increment the index of a loop and must not be used to compute the result itself.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

III.0.2 Instructions

The prototype of the function to write is the following:

```
fn adder(a: u32, b: u32) -> u32;
```

Chapter IV

Exercise 01 - Multiplier

IV.0.1 Goal

The goal is the same as the previous exercise, except the returned natural number equals `a * b`. The only operations you're allowed to use are:

- `&` (bitwise AND)
- `|` (bitwise OR)
- `^` (bitwise XOR)
- `<<` (left shift)
- `>>` (right shift)
- `=` (assignment)
- `==`, `!=`, `<`, `>`, `<=`, `>=` (comparison operators)

The incrementation operator (`++` or `+= 1`) is allowed **only** to increment the index of a loop and must not be used to compute the result itself.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

IV.0.2 Instructions

The prototype of the function to write is the following:

```
fn multiplier(a: u32, b: u32) -> u32;
```

Chapter V

Exercise 02 - Gray code

V.0.1 Goal

You must write a function that takes an integer `n` and returns its equivalent in Gray code.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

V.0.2 Instructions

The prototype of the function to write is the following:

```
fn gray_code(n: u32) -> u32;
```

Examples:

```
println!("{}", gray_code(0));
// 0
println!("{}", gray_code(1));
// 1
println!("{}", gray_code(2));
// 3
println!("{}", gray_code(3));
// 2
println!("{}", gray_code(4));
// 6
println!("{}", gray_code(5));
// 7
println!("{}", gray_code(6));
// 5
println!("{}", gray_code(7));
```

```
// 4
println!("{}", gray_code(8));
// 12
```

Chapter VI

Exercise 03 - Boolean evaluation

VI.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation, evaluates this formula, then returns the result.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

VI.0.2 Instructions

Each character represents a symbol. The input contains only the following characters:

Symbol	Mathematical equivalent	Description
0	$\perp$	false
1	$\top$	true
!	$\neg$	Negation
&	$\wedge$	Conjunction
|	$\vee$	Disjunction
ˆ	$\oplus$	Exclusive disjunction
>	$\Rightarrow$	Material condition
=	$\Leftrightarrow$	Logical equivalence

Here are some examples of propositional formulas:

```
10&
```

Equivalent:

$$\top \wedge \perp$$

```
10|
```

Equivalent:

$$\top \lor \bot$$

```
10|1&
```

Equivalent:

$$(\top \lor \bot) \land \top$$

```
101|&
```

Equivalent:

$$\top \land (\bot \lor \top)$$

If the formula is invalid, the behaviour of the return value is undefined (but we encourage you to print an error message).

The prototype of the function to write is the following:

```
fn eval_formula(formula: &str) -> bool;
```

Examples:

```
println!("{}", eval_formula("10&"));
// false
println!("{}", eval_formula("10|"));
// true
println!("{}", eval_formula("11>"));
// true
println!("{}", eval_formula("10="));
// false
println!("{}", eval_formula("1011||="));
// true
```

Reverse polish notation, although easy to parse and evaluate for a computer, is very ugly and difficult to read for humans. How about using a (binary) Abstract Syntax Tree structure to parse and model your formula? You can probably add interesting utils, like converting this binary tree to a regular tree, or ways to visualize it. You can also use one of your language's existing tree libraries. Doing this now could probably prove useful to your understanding for the following exercises!
As an example, the formula $\top \land (\bot \lor \top)$ can be represented by the following tree:

```
   ∧
  / \
 ⊤   ∨
    / \
   ⊥   ⊤
```

Chapter VII

Exercise 04 - Truth table

VII.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation, and writes its truth table on the standard output.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

VII.0.2 Instructions

Each character represents a symbol. The input contains only the following characters:

Symbol	Mathematical equivalent	Description
A...Z	$A..Z$	Distinct variables with unknown values
!	$\neg$	Negation
&	$\land$	Conjunction
|	$\lor$	Disjunction
ˆ	$\oplus$	Exclusive disjunction
>	$\Rightarrow$	Material condition
=	$\Leftrightarrow$	Logical equivalence

A formula can have up to 26 distinct variables, one per letter. Each variable can be used several times.

As an example, for the following formula:

```
AB&C|
```

which is equivalent to:

$$(A \land B) \lor C$$

the function shall write the following to the standard output:

```
$ ./ex04 | cat -e
| A | B | C | = |$
|---|---|---|---|$
| 0 | 0 | 0 | 0 |$
| 0 | 0 | 1 | 1 |$
| 0 | 1 | 0 | 0 |$
| 0 | 1 | 1 | 1 |$
| 1 | 0 | 0 | 0 |$
| 1 | 0 | 1 | 1 |$
| 1 | 1 | 0 | 1 |$
| 1 | 1 | 1 | 1 |$
```

If the formula is invalid, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function to write is the following:

```
fn print_truth_table(formula: &str);
```

Chapter VIII

Interlude 00 - Rewrite rules

Rewrite rules allow us to transform a logical expression into an equivalent expression. All logic systems have their own sets of rules that are allowed and you're supposed to follow (boolean algebra is just one of many!). In boolean algebra, the following rules exist (non-exhaustive):

VIII.1 Elimination of double negation

$$(\neg\neg A) \Leftrightarrow A$$

VIII.2 Material conditions

$$(A \Rightarrow B) \Leftrightarrow (\neg A \lor B)$$

VIII.3 Equivalence

$$(A \Leftrightarrow B) \Leftrightarrow ((A \Rightarrow B) \land (B \Rightarrow A))$$

VIII.4 De Morgan's laws

$$\neg(A \lor B) \Leftrightarrow (\neg A \land \neg B)$$
$$\neg(A \land B) \Leftrightarrow (\neg A \lor \neg B)$$

VIII.5 Distributivity

$$(A \land (B \lor C)) \Leftrightarrow ((A \land B) \lor (A \land C))$$
$$(A \lor (B \land C)) \Leftrightarrow ((A \lor B) \land (A \lor C))$$

⚠ `Watch out, unlike the one-way distributivity for` × `over` +`, this one goes both ways!`

Chapter IX

Exercise 05 - Negation Normal Form

IX.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation, and returns an equivalent formula in **Negation Normal Form** (NNF), meaning that every negation operators must be located right after a variable.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

IX.0.2 Instructions

The format of the propositional formulas is the same as the previous exercise.

For example, if the function takes `AB&!` (equivalent: $\neg(A \land B)$) in input, it should return `A!B!|` (equivalent: $\neg A \lor \neg B$) as output.

The result must only contain variables and the following symbols: `!`, `&` and `|` (even if the input contains other operations).

There may be several valid results, only one is required.

If the formula is invalid, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function to write is the following:

```
fn negation_normal_form(formula: &str) -> String;
```

Examples:

```
println!("{}", negation_normal_form("AB&!"));
// A!B!|
println!("{}", negation_normal_form("AB|!"));
// A!B!&
println!("{}", negation_normal_form("AB>"));
// A!B|
println!("{}", negation_normal_form("AB="));
// AB&A!B!&|
println!("{}", negation_normal_form("AB|C&!"));
// A!B!&C!|
```

Chapter X

Exercise 06 - Conjunctive Normal Form

X.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation, and returns an equivalent formula in **Conjunctive Normal Form** (CNF). This means that in the output, every negation must be located right after a variable and every conjunction must be located at the end of the formula.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

`You may use the function you wrote during the previous exercise to help you.`

X.0.2 Instructions

The format of the propositional formulas is the same as the previous exercise.

For example, if the function takes `ABCD&|&` (equivalent: $A \land (B \lor (C \land D))$) in input, it should return `ABC|BD|&&` (equivalent: $A \land (B \lor C) \land (B \lor D)$) as output.

The result must only contain variables and the following symbols: `!`, `&` and `|` (even if the input contains other operations).

There may be several valid results, only one is required.

The size of the output formula may grow exponentially with the size of the input. There exist an algorithm to avoid this but it isn't required to use it.

If the formula is invalid, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function to write is the following:

```
fn conjunctive_normal_form(formula: &str) -> String;
```

Examples:

```
println!("{}", conjunctive_normal_form("AB&!"));
// A!B!|
println!("{}", conjunctive_normal_form("AB|!"));
// A!B!&
println!("{}", conjunctive_normal_form("AB|C&"));
// AB|C&
println!("{}", conjunctive_normal_form("AB|C|D|"));
// ABCD|||
println!("{}", conjunctive_normal_form("AB&C&D&"));
// ABCD&&&
println!("{}", conjunctive_normal_form("AB&!C!|"));
// A!B!C!||
println!("{}", conjunctive_normal_form("AB|!C!&"));
// A!B!C!&&
```

Not all valid CNF expressions for a given formula are created equal. There is a way to simplify a CNF using something called a Karnaugh map. If you wish, you can implement your function so that it returns a simplified CNF. (This is noteworthy for those interested in the design of logical electronic circuits.)

Chapter XI

Exercise 07 - SAT

XI.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation and tells whether it is satisfiable.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

You may use the functions you wrote from the previous exercises to help you.

XI.0.2 Instructions

The format of the propositional formulas is the same as the previous exercise.

The function has to determine if there is at least one combination of values for each variable of the given formula that makes the result be $\top$. If such a combination exists, the function returns `true`, otherwise, it returns `false`.

If the formula is invalid, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function to write is the following:

```
fn sat(formula: &str) -> bool;
```

Examples:

```
println!("{}", sat("AB|"));
// true
println!("{}", sat("AB&"));
// true
println!("{}", sat("AA!&"));
// false
println!("{}", sat("AA^"));
// false
```

Chapter XII

Interlude 01 - Rules of inference

A rule of inference is a notation which takes premises (propositional formulas), and deduces conclusions about them.

Here is an example of a rule of inference.

$$\frac{A \Rightarrow B, A}{\therefore B}$$

The above example is called **Modus Ponens** and can be read:

- $A \Rightarrow B$: "A implies B"; "If A, then B"
- A: "A (is true)"
- $\therefore B$: "Therefore B (is true)"

The **Modus Tollens** also holds:

$$\frac{A \Rightarrow B, \neg B}{\therefore \neg A}$$

Both the Modus Ponens and Modus Tollens are fundamental rules in mathematical reasoning and are very useful in type theory for type inference (they are notably used in compilers to check a program's typing).

XII.1 Reductio ad absurdum

Reductio ad absurdum, (from Latin: "reduction to absurdity") is a way of proving a result by trying to prove its opposite.

If trying to prove that "B isn't true" results in a contradiction, then B must be true. Formally:

$$\frac{(A \cup \neg B) \Rightarrow \bot}{\therefore A \Rightarrow B}$$

where A is the set of assertions that are already known to be true and B is the proposition to prove.

This rule is based off a principle known as the **Law of excluded middle**, which states that for every proposition P, if P is false, then $\neg P$ must be true.

Chapter XIII

Interlude 02 - Set theory

XIII.1 Naive Set Theory

Set Theory is the study of mathematical objects called **sets**. A set is collection of objects that can be of any kinds (numbers, function, other sets, potatoes...) but a set cannot contain the same object twice, nor contain itself (for a reason seen below). There do exist sets of sets, though. You can see a set as an array that can be finite or infinite, and cannot be indexed (the order of the objects doesn't matter, only whether an element is in the set or not matters).

This theory appeared around the 1870s. It originated from the works of Richard Dedekind and Georg Cantor and is used as a foundation for the whole of mathematics.

XIII.2 Russell's Paradox

In 1901, Bertrand Russell discovered what is today known as **Russell's Paradox** which shows that "naive" set theory was flawed.

Imagine a set A that contains every sets that do not contain themselves. If A doesn't contain itself, then it must contain itself. But if it contains itself, it must not contain itself...

Formally:

$$A = \{x \mid x \notin x\} \Rightarrow (A \in A \iff A \notin A)$$

This paradox led to the creation of other, more adavanced theories such as **Zermelo-Fraenkel (ZF) set theory**, **category theory** and **type theory** (this one is very useful in computer science).

Coming back to naive set theory (which is more of a collection of theories than a single theory), even though is has some paradoxes, it remains useful in many cases (especially when comparing different versions of set theory).

XIII.3 Symbols

Set theory introduces new symbols, including (non-exhaustive):

- $\emptyset$: The empty set, the set with no elements.
- $a \in B$: "a in B" (it is true that a is an element of the set B)
- $a \notin B$: "a not in B"
- $\forall a \in B$: "For all a in B"" (the proposition is true for every value a in B)
- $\exists a \in B$: "There exists a in B" (the proposition is true for at least one value a in B)
- $A \subseteq B$: A is a subset of B (all the elements of A are in B, and A may be equal to B itself)
- $A \not\subseteq B$: A is not a subset of B (some elements of A are not in B)
- $A \supseteq B$: A is a superset of B (all the elements of B are in A, and A may be equal to B itself)
- $A \not\supseteq B$: A is not a superset of B (some elements of B are not in A)

The number of elements in a set A is called its **cardinal**. Often denoted $card(A)$ or $|A|$. If a set is infinite, its cardinal is an infinite number. Note that there are multiple, distinct, infinite cardinals.

XIII.4 Link between Logic and Set Theory

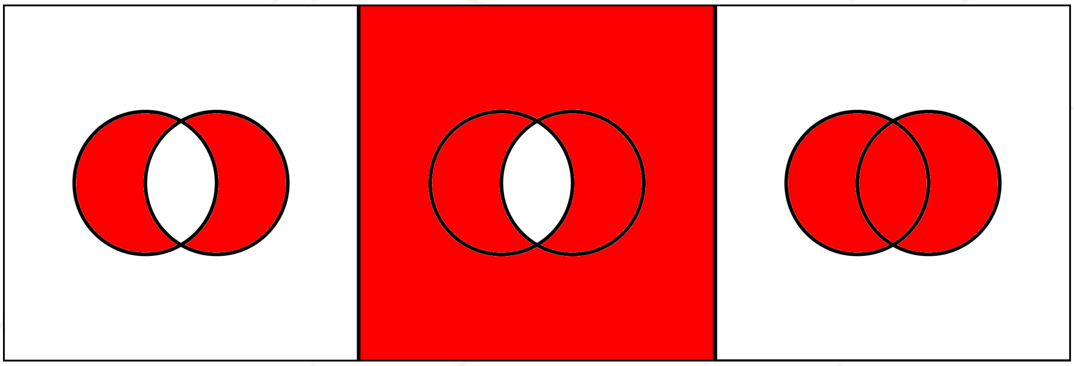
Actually, set theory is based on logic itself; or maybe it's the other way around, and you can define logic through set theory! Anyways, many concepts between the two are equivalent/synonymous. (Pretty cool, isn't it?). The following logical operations have a counterpart in set theory (they are not the only operations that exist though):

Boolean Algebra	Boolean Algebra name	Set Theory	Set Theory name
$\neg A$	Negation	$A^\complement$ or $\overline{A}$	Complement
$A \lor B$	Disjunction	$A \cup B$	Union
$A \land B$	Conjunction	$A \cap B$	Intersection

Most of the other operations can be built on top of the complement and the union (yes, even the intersection can). The rewrite rules that we mentioned earlier (double negation, De Morgan's Laws, ...) also work on sets.

XIII.5 Venn diagrams

You've probably seen a Venn diagram before. They are used to express both logical operations, and operations on sets.



These three diagrams represent operations on two sets, each set represented by a circle (in a globally encompassing set which is a rectangle). The result of the operation being the colored part. **Can you guess which operations they are?**

Chapter XIV

Exercise 08 - Powerset

XIV.0.1 Goal

You must write a function that takes as input a set of integers, and returns its powerset.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XIV.0.2 Instructions

Let A be a set. Let $B = \mathcal{P}(A)$ be the powerset of A.

The function must take the set A as input and return the set B.

It is assumed that the set passed as parameter is valid (no duplicate elements).

The prototype of the function to write is the following:

```
fn powerset(set: Vec<i32>) -> Vec<Vec<i32>>;
```

Chapter XV

Exercise 09 - Set evaluation

XV.0.1 Goal

You must write a function that takes as input a string that contains a propositional formula in reverse polish notation, and a list of sets (each containing numbers), then evaluates this list and returns the resulting set.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XV.0.2 Instructions

Each character represents a symbol. The input contains only the following characters:

Symbol	Mathematical equivalent	Description
A...Z	$A..Z$	Distinct sets
!	$\neg$	Negation
&	$\wedge$	Conjunction
|	$\vee$	Disjunction
ˆ	$\oplus$	Exclusive disjunction
>	$\Rightarrow$	Material condition
=	$\Leftrightarrow$	Logical equivalence

Each letter represents a set that is passed to the function. The set A is the first set, the set B is the second set, etc...

⚠ The globally encompassing set is considered to be the union of all the sets given as parameters.

If the formula is invalid, or if the amount of sets provided in the list is not equal to the amount of variables in the formula, the behaviour is undefined (however, we encourage you to print an appropriate error message).

The prototype of the function to write is the following:

```
fn eval_set(formula: &str, sets: Vec<Vec<i32>>) -> Vec<i32>;
```

Examples:

```
let sets = {
    {0, 1, 2},
    {0, 3, 4},
};
let result = eval_set("AB&", &sets);
// [0]

let sets = {
    {0, 1, 2},
    {3, 4, 5},
};
let result = eval_set("AB|", &sets);
// [0, 1, 2, 3, 4, 5]

let sets = {
    {0, 1, 2},
};
let result = eval_set("A!", &sets);
// []
```

Remember: The order of the elements in the resulting set doesn't matter.

Chapter XVI

Interlude 03 - Space Filling Curves

XVI.1 Common sets

We previously talked about naive set theory. But we didn't talk about the sets commonly used in algrebra, arithmetic and analysis (non-exhaustive):

Symbol	Name	Closest type in the C language	Notes
$\mathbb{N}$	Natural numbers	unsigned int	$\mathbb{N}$ includes zero, whereas $\mathbb{N}^*$ doesn't
$\mathbb{Z}$	Integers	int	
$\mathbb{Q}$	Rational numbers	float	Represents the set of numbers that can be expressed as the fraction of two integers: $\mathbb{Q} = \{\frac{a}{b} \mid a \in \mathbb{Z},\ b \in (\mathbb{Z} \setminus \{0\})\}$
$\mathbb{R}$	Real numbers	float	
$\mathbb{C}$	Complex numbers	complex	A complex number can be represented by $a + bi$ where $i^2 = -1$

There's a few differences between mathematics and C types, for example (non-exhaustive):

- int can overflow but $\mathbb{N}$ and $\mathbb{Z}$ cannot. Technically, **intX**, where X is the number of bits, corresponds to the space $\mathbb{Z}/(2^X)\mathbb{Z}$ ("ring of the integers modulo 2^X"), which acts sort of like a clock where there are 2^X hours and you can only land on hours.
- float has a finite precision (it is a set of specific binary rational values, used to approximate other values) whereas $\mathbb{Q}$ and $\mathbb{R}$ have an infinite precision.

The following holds:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$$

XVI.2 A mathematical structure: Groups

A group is a structure (you can see this as the "environment" in which you are working) that has a set and an internal binary operation.

An **internal binary operation** (also a "stable binary operation" or "closed binary operator") is an operation that takes two elements of the set and returns an element that is still in the same set. Whereas an **external binary operation** is an operation that takes two elements of the set and returns an element in **another** set.

Examples of binary operations are + (addition) or × (multiplication). Though one must be careful as to the set for which these are defined! For example, − (subtraction) is not closed on $\mathbb{N}$, but it is closed on $\mathbb{Z}$.

A group is declared as follows:

$$(G, \cdot)$$

where G is a set and $\cdot$ is a binary operation.

A group has to respect the following properties:

- **Internal Binary Operation**: $\forall a, b \in G, (a \cdot b) \in G$ (for all a and b in G, the result of $a \cdot b$ is still in G)
- **Associativity**: $\forall (a, b, c) \in G^3, a \cdot (b \cdot c) = (a \cdot b) \cdot c$ (the order of the parenthesis doesn't matter)
- **Identity element**: $\exists e \in G, \forall a \in G, e \cdot a = a \cdot e = a$ (there exists an element e, called the identity (or unit, or more rarely, neutral), that has no effects on the others, for the given operation)
- **Symmetry**: $\forall a \in G, \exists b \in G, a \cdot b = b \cdot a = e$, where e is the identity element (for all elements in G, there exists an inverse element)

If one of the above is not respected, the structure is not a group.

For example, the following are groups:

- $(\mathbb{Z}, +)$: The additive group of integers

- $(\mathbb{R} \setminus \{0\}, \times)$: The multiplicative group of real numbers (excluding 0)

Can you tell what is their respective identity element?

And the following are not groups:

- $(\mathbb{Z}, \times)$
- $(\mathbb{R}, \times)$

Can you figure out why?

An abelian group (or commutative group) is a group which fullfills one more condition:

- **Commutativity**: $\forall (a, b) \in G^2, a \cdot b = b \cdot a$ (the order of the operands doesn't matter)

Thus, the above groups $(\mathbb{Z}, +)$ and $(\mathbb{R} \setminus \{0\}, \times)$ are also both abelian.

The **order** of a group is the **cardinal** of its set

For more informations about groups, watch:

- Euler's formula with introductory group theory
- Group theory, abstraction, and the 196,883-dimensional monster

Pheww! I know, this might seem a bit confusing if you're beginning but you have to remember that this is very abstract. Examples and counter-example are very helpful, and they abound for terms like "group", so it's very important vocabulary in the long run, since it makes things a lot easier. Knowing something is a "group" is a quick and easy way of communicating what you're allowed to do with a certain set + operation combo.

If this is too hard for you for now, you can just ignore the words "structure" and "group" in the next section and come back to this later.

XVI.3 Morphisms

A morphism is the abstraction of a function. Basically, a morphism is a mapping of values from one structure to a structure of the same type.

A morphism is said to preserve the structure because if the source structure is a group, the target set of a group morphism will also be a group (but not necessarily the same group).

A morphism f that maps values from set A to set B is declared as follows:

$$f : A \to B$$

If the morphism takes two arguments (or a vector in two dimensions) it is declared as follows:

$$f : A^2 \to B$$

The **composition operator** can be applied to two morphisms. It is defined as follows:

$$(f \circ g)(x) = f(g(x))$$

where f and g are two morphisms.

If we consider the "collection of all sets" (something called a category in category theory, you can think of categories as programming types), you will most often see the category Set, for which the morphisms are functions. I say "most often", since you can define another choice of morphisms to turn the "collection of all sets" into a category.

These "set functions" have important properties they can verify, or not. We say a set function is:

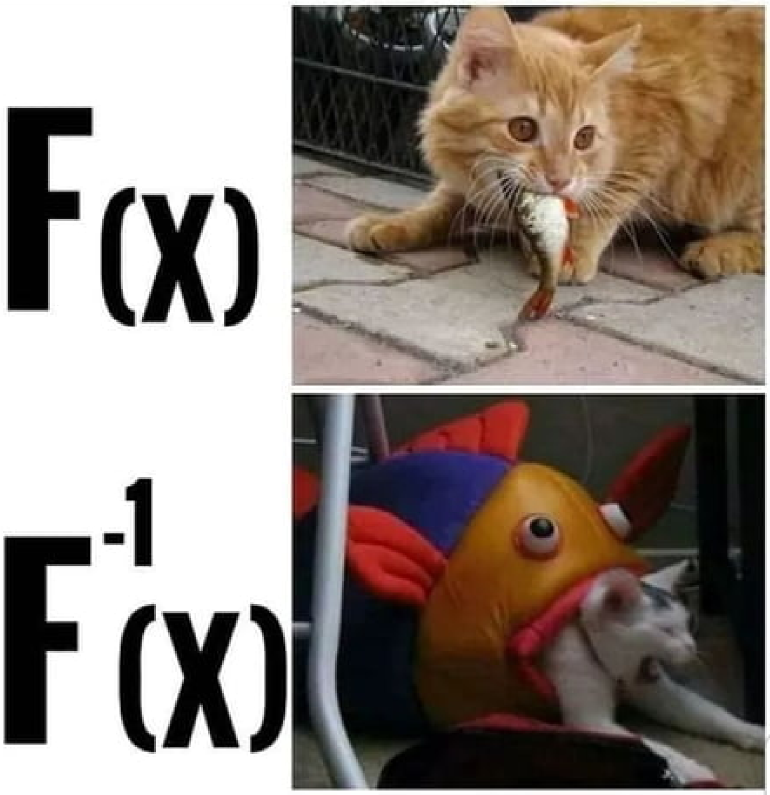
- **injective**: if every element from the source set (domain) is mapped to **exactly one** element in the target set (codomain)
- **surjective**: if every element in the target set (codomain) is mapped from **at least one** element in the source set (domain)
- **bijective**: if it is both injective and surjective, meaning that each elements in the source and target are mapped exactly 1-to-1

In general category theory, these notions, which are very important for set theory, generalize respectively to those of "monomorphism", "epimorphism" and "isomorphism".

If a morphism f is bijective, there exists an inverse morphism f^{-1} such that:

$$(f^{-1} \circ f)(x) = (f \circ f^{-1})(x) = id_X(x) = x$$

Where $x \in X$ and id_X is the identity morphism over the set X. Notice that id_X is the identity element in groups of set functions, and what you see above is the symmetry property for a group of function with the $\circ$ operation.



A little meme to keep your attention so far :D

There exist much extra vocabulary surrounding morphisms such as **automorphisms**, **endomorphisms**, **isomorphisms**, etc...

For more informations, you can read: https://en.wikipedia.org/wiki/Morphism.

XVI.4 Space-Filling Curves

A space-filling curve is a continuous curve which maps a closed interval $[0; 1] \in \mathbb{R}$ to a set of values in 1 or more dimensions, so as to cover the whole space.

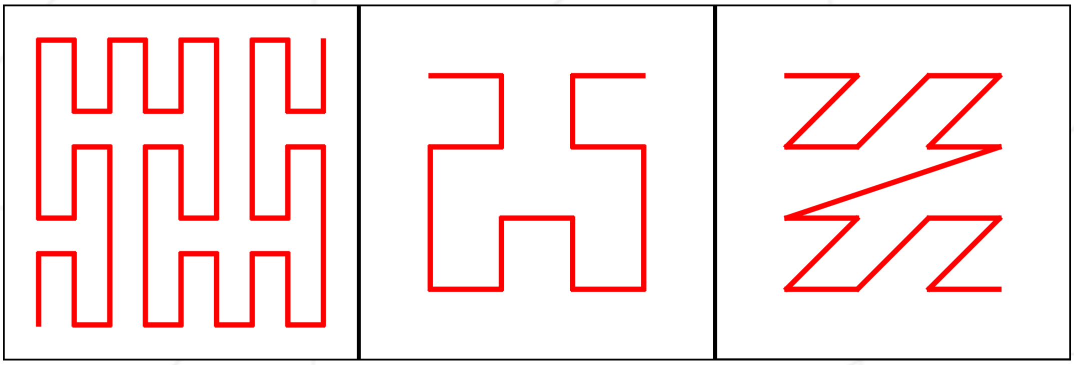
A space-filling curve is defined as:

$$f : [0; 1] \in \mathbb{R} \to \mathbb{M}^n$$

Where $[0, 1]$ is the source set (it contains values from 0 to 1, such as 0.543543 and 0.3333...), and $n \in \mathbb{N}$ is the number of dimensions for a manifold $\mathbb{M}$ (a geometric shape or space) that we wish to cover.

It's important to note that f **must** be bijective.

Space-filling curves are defined by a number of iterations. The more iterations, the more precise it is (the more space it covers).



Second iteration of 3 different space filling curves, from left to right: the **Peano curve**, the **Hilbert curve** and the **Lebesgue curve** (also called **Z-order curve**)

The Z-order curve in particular is very important since it's used a lot in computer science. Basically, by taking the inverse function of a space-filling curve, you can "store a whole space inside a line". For example, some GPUs use this curve to store textures in memory and decrease (memory) cache misses.

`Look up what does continuity means in the context of a space filling curve`

Chapter XVII

Exercise 10 - Curve

XVII.0.1 Goal

You must write a function (the inverse of a space-filling curve, used to encode spatial data into a line) that takes a pair of coordinates in two dimensions and assigns a unique value in the closed interval $[0;1] \in \mathbb{R}$.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XVII.0.2 Instructions

Let f be a function and let A be a set such as:

$$f : (x, y) \in [[0; 2^{16} - 1]]^2 \subset \mathbb{N}^2 \to A$$

$$A \subset [0; 1] \subset R$$

Where:

$$card(A) = card([[0; 2^{16} - 1]]^2)$$

The above function f is bijective and represents the function to implement. You're free to use the method you want as long as it stays bijective.

If the input is out of range, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function is the following:

```
fn map(x: u16, y: u16) -> f64;
```

ⓘ This exercise is not mandatory.

Chapter XVIII

Exercise 11 - Inverse function

XVIII.0.1 Goal

You must write the inverse function f^{-1} of the function f from the previous exercise (so this time, this is a space-filling curve, used to decode data from a line into a space).

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XVIII.0.2 Instructions

Let f and A be the function and set from the previous exercise, then:

$$f^{-1} : A \to [[0; 2^{16} - 1]]^2 \subset \mathbb{N}^2$$

The above function must be implemented such that the following expressions are true for values that are in range:

$$(f^{-1} \circ f)(x, y) = (x, y)$$
$$(f \circ f^{-1})(x) = x$$

If the input is out of range, the behaviour is undefined (but we encourage you to print an error message).

The prototype of the function is the following:

```
fn reverse_map(n: f64) -> (u16, u16);
```

ℹ This exercise is not mandatory.

Chapter XIX

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.