MarkdownToLaTeX: Mathsheet

Gabriel Cordier

June 23, 2023

Contents

Chapter 1

Why formal specifications matter (biographical note)

To briefly describe what (formal) specification is, I will stand on Leslie Lamport's definition [4]:

"A specification does not describe the correct behavior of a system, rather it describes a universe in which the system and its environment are behaving correctly."

To see this, let us consider the Earth (E) orbiting around the Sun (S) and assume that we need to *predict* the position of the Earth (with respect to the Sun) at given time t. All we have is: Newton's second law, Newton's law of universal gravitation, and some basic knowledge about differential equations.

Our first level of abstraction consists in identifying each celestial bodies with its center of mass. We make the customary (yet redundant) assumption that the Earth's orbit lies in the complex plane (with the Sun at the origin), so that the Earth's position is a complex number. The smooth function $\mathrm{p} : \mathbf{R} \to \mathbf{C}, \mathrm{t} \mapsto \mathrm{p}(\mathrm{t})$ now maps time t to $\mathrm{p}(\mathrm{t})$ the position of the Earth at time t. First order derivative $\dot{\mathrm{p}} = \mathrm{dp}/\mathrm{dt}$ is then the Earth's velocity, as second order derivative $\ddot{\mathrm{p}} = \mathrm{d}^2\mathrm{p}/\mathrm{dt}^2$ is the Earth acceleration. Finally, we assume without loss of generality that our current time is $\mathrm{t} = 0$, so that $\mathrm{p}(0) = \mathrm{R} > 0$ is our initial position.

Combined with Newton's law of universal gravitation (which provides the gravitational constant $\mathrm{G} > 0$), Newton's second law leads to

$$\ddot{\mathrm{p}} + \mathrm{GM_S}\frac{\mathrm{p}}{|\mathrm{p}|^3} = 0; \tag{1.1}$$

where $\mathrm{M_S}$ is the Sun's mass. Let's be humble: The problem is crazy hard. Additionally, it is ill-founded, since we have no assumption about velocity $\dot{\mathrm{p}}(0)$. We may discard the trivial case "$\mathrm{p}(0)$ and $\dot{\mathrm{p}}(0)$ are aligned" (for instance, if the Earth falls into the Sun), but this is of no help: We are not living in such case. Hopefully, we empirically know that $|\mathrm{p}(\mathrm{t})|$ does not vary much, hence the reasonable assumption $|\mathrm{p}| = \mathrm{R}$. Our abstraction now turns out to be

$$\ddot{\mathrm{p}} + \mathrm{Cp} = 0, \tag{1.2}$$

as $\mathrm{C} = \mathrm{GM_S}/\mathrm{R}^3$. So,

$$\mathrm{p}(\mathrm{t}) = \mathrm{R}e^{\pm i\sqrt{\mathrm{C}}\mathrm{t}} \quad (\mathrm{t} \in \mathbf{R}). \tag{1.3}$$

In other words, The Earth has a circular orbit of radius R, the Earth velocity remains constant in magnitude and the whole run has period $\mathrm{T} = 2\pi/\sqrt{\mathrm{C}} \approx 374$ days - a $2,5\%$ error, then. If this is sufficiently precise, then we reach a [class of] universe[s] in which the Earth and its environment (time and Sun) behave correctly:

(*a*) A universe $\mathrm{U_c}$, in which the Earth orbits clockwise;

(*b*) A universe $\mathrm{U_{cc}}$, in which the Earth orbits counterclockwise;

(*c*) Two other universes $\mathrm{U_{-c}}, \mathrm{U_{-cc}}$, in which times goes backward.

Remark that our model does not decide the universe we are living in. Any given $\dot{\mathrm{p}}(0)$ would divide our options by two, but our scope statement aimed at generality... It is true that our solutions for $\mathrm{p(t)}$ are easy to grasp and look accurate, but *only under the rule of thumb* $\mathrm{p(t)} \simeq \mathrm{R}$, and *as long as a* $2,5\%$ *error is an acceptable number.* The point is: Even when we consider two simple objects (our points, equipped with mass attributes) ruled by two elegant invariants (principles), complexity rises to surface as we start looking at how those two components interact each other. Furthermore, choosing the right level of abstraction makes models false - but this is fine as long as we know why.

I started to get interested in formal methods since I believed that a mathematical way of thinking could help in software development: Not only to find aggresive optimizations but also to deal with problem boundaries. Clémenceau said that "*war is too important a matter to be left to the military*". I think he was right. Sofware design should not be left to "coders".

There is a psychological factor: "Coders" may not like formal methods when they believe they are sufficiently smart to not waste time with them. It is sad to observe how smart people can believe that Laplacian worlds [8] are effective. In my humble opinion, being smart is *asymptotically* of no help: As time goes, any system complexity stops plateauing and there is always a moment at which no one one on Earth can have a perfect recall of all the system behaviors. In short,

Being smarter only means that you will fail later.

Note how glossary does matters:

- A coder is someone who encodes, *i.e.* who turns plain into cipher;
- A developer is someone who develops, *i.e.* who expands something that already exists;
- On the other hand, a programmer is someone who programs, *i.e.* someone who creates, designs, expects and proves.

To be honest, my point of view is biased. I have studied math and that has probably shaped my mind for ever. Unless I started to study math due to a prior taste for abstraction. Anyway, I firmly believe that 90% of bad code is the consequence of bad management or commercial pressure, but I also believe that sometimes the right mindset is not here. One possible source of trouble is that human beings tend to identify object definition with object function. For instance: A shoe, a door, a road, a bridge, ... Unfortunately for them,

- Such description is not the object itself, only a "high-level" representation. As an example, it is easy to picture what a bridge is, but bridges may nest very complex feats of civil engineering.

- In contrast, how the very components interact each other is often underestimated: Behaviors that are not part of the system description remain in the dark. To see that, let us play billiard with Laplace [8]: Putting the balls in the rack is simple, being good at the game is hard.

One may object that billiard balls behavior is not discrete. This is a fair objection, so let us take a card deck, which is a simple object: It is hard to be good at pocker. Note that I am not putting anything new: Cybertenics people already distinguished *functional complexity* from *structural complexity*.

Another objection (which is actually a variant of the previous "smart-enough" one) may be stated as 'Let's stick to the dev: We have software crafmanship". Craftmanship is an intrinsically positive value: More craftmanship is better than less craftmanship. But if craftmanship means a combination of pragmatism and expertise that opposes conceptual thinking, then such "crafmanship" is irrelevant here: It would be like entering a civil engineering design studio and claim "People, drop the finite elements [7], we have crafmanship!". For the record, before finite elements came along, bridges used to collapse for ages. You may think I am trying some *reductio ad absurdum* so let us consider the famous Florence Dome: It was built (not to mention the design) from methods that *3000 years of craftmanship could not unveil*. Closer to us, the first Xbox: By writing its memory coherence protocol in a specification language (TLA+; see next paragraph), a Microsoft team (Chuck Thacker and an intern) found a subtle but obnoxious bug [3] just before the Christmas launch. IBM the manufacturer acknowledged that *none of their regression tests could unveil the bug*.

I have already seen bad implementation and some of them were mine. As I keep writing programs, the longer time goes, the slower I am. Steps by steps, **I am starting to think hard before I write any piece of code**. The process is not achieved. I am writing MarkdownToLaTeX fueled with the ideas from Leslie Lamport's *Specifying Sytems* [5]. Actually, MarkdownToLaTeX began as a toy project in which I could implement TLA (Temporal Logic of Actions), the logic that *Specifying Systems* is about. The MarkdownToLaTeX parsing implements a state machine whose steps are decided by (1) the last input character, and (2) a bounded memory that stores the necessary context. The state machine was first written in the specification language TLA+ (as TLA is the logic underlying TLA+) then tested within the TLA Toolbox [1]. The whole sourcecode was written in Python.

By the way, does MarkdownToLaTeX work? The PDF you are reading now has been made with MarkdownToLaTeX, its source code embeds some Markdown. So, to some extent: Yes.

Chapter 2

Fundamental objects

2.1 Introduction

In this chapter and the two following ones, we introduce the basic necessary mathematical concepts, and bind them to programming or Python features.

We expect the reader to be familiar (through intuition or formal training) with the "basic usual math", formally the math that is founded by ZFC set theory. ZFC embedds all usual material: sets, ordered pairs, union, intersection, tuples, functions, natural numbers, and so on.

The whole content from chapters 1, 2, 3 may be read as a reminder, not a crash course.

There is absolutely no original work here, even if explicitely connecting associativity of Cartesian product with concatenation was my personal touch. Here are two references for further reading: [6][2]

2.2 Logical symbols

2.2.1 Equalities

$\triangleq$ is a specialization of $=$ We say that x $\triangleq$ y if and only if (from now on denoted by **iff**) x and y are assumed to be equal. Usually x $\triangleq$ y means that x is assigned the previously known value y (some authors write it x := y) but this is not a limitation. Definitions can be redundant and may overlap. The only restriction is that x $\triangleq$ y is inconsistent whether x $\neq$ y.

2.2.2 Logical connectors

TODO

2.3 Sets

2.3.1 ZFC

In set theory, the primitive (fundamental) notion is the notion of set. Sets and operations over sets are meant to capture what we usually mean by "set" or"collection", or "x belongs to X", *e.g.* "Bart is a member of the family, among Homer, Liza, Maggie, and Marge", hence the naming "set". So, in ZFC set theory, everything (almost) is a set. Of course we tend to ignore it when we deal with "high-level" objects, such as real numbers.

2.3.2 Empty set

The empty set $\emptyset$ is the set that has no element. Following the context, it may be denoted by 0, `False`, ( ) the empty list, [ ] the empty matrix, ε the empty word.

2.4 Numbers

In ZFC, the axiom of infinity more or less implicitely asserts the existence of the natural numbers. Actually, it seems to be really hard to do anything without prior intuition of plurality and space. Bear in mind that what you are reading now is a collection of symbols and that you are reading them from left to right.... But, formally, in ZFC, natural numbers are defined from the empty set $\emptyset$, as follows

$$0 \triangleq \emptyset \tag{2.1}$$
$$1 \triangleq \{0\} = \{\emptyset\} \tag{2.2}$$
$$2 \triangleq \{0,1\} = \{\emptyset, \{\emptyset\}\} \tag{2.3}$$
$$\vdots$$
$$\mathrm{n}+1 \triangleq \{0,\dots,\mathrm{n}\} \tag{2.4}$$

Note that $0 \subseteq 1 \subseteq 2 \subseteq \cdots$, what we usually write $0 \leq 1 \leq 2 \leq \cdots$. The strict version of $\leq$ is $<$, *i.e.* $\mathrm{x} < \mathrm{y}$ is $\mathrm{x} \leq \mathrm{y}$ with the extra information that $\mathrm{x} \neq \mathrm{y}$. The binary relations (see Cartesian product) $\geq$ and $>$ are nothing but $\leq, <$ read from right to left. The set of all natural numbers is denoted by $\mathbf{N}$. Addition $+$ over $\mathbf{N}$ is extended as follows,

$$\mathrm{n}+0 \triangleq \mathrm{n} \tag{2.5}$$
$$\mathrm{n}+(\mathrm{k}+1) \triangleq (\mathrm{n}+\mathrm{k})+1 \quad (\mathrm{n} \in \mathbf{N}, \mathrm{k} \in \mathbf{N}). \tag{2.6}$$

$\mathbf{N}$ can be embedded in $\mathbf{Z} = \{\dots,\text{-}2,\text{-}1,0,1,2,\dots\}$, the set of all integers. $\mathbf{Z}$ is equipped with a (commutative) generalized addition $+$ that can be reversed through substraction $-$ (in the sense that $\mathrm{n}-\mathrm{n} = \mathrm{n}+(-\mathrm{n}) = 0$). Multiplication $\times$ and Euclidian division derive from $\mathbf{Z}$'s algebra. The whole construction is a long chain of formal computations (we do not digg into details) whose output is the arithmetic. The set of all nonnegative integers $\{0,1,2,\dots\}$ is (identified with) $\mathbf{N}$, so that

$$\mathbf{N} \triangleq \{0,1,2,\dots\} \tag{2.7}$$
$$\mathbf{Z}_+ \triangleq \{1,2,\dots\} \triangleq \mathbf{N}_+ \tag{2.8}$$
$$\tag{2.9}$$

Rational numbers are defined once $\mathbf{Z}$ is constructed, still by arithmetic means. The real line $\mathbf{R}$ is constructed from the rational numbers.

2.5 Cartesian product

Given a set X and a set Y, the *Cartesian product* $\mathrm{X} \times \mathrm{Y}$ is primarily defined as the set of all ordered pair(s) (x,y) $(\mathrm{x} \in \mathrm{x}, \mathrm{y} \in \mathrm{Y})$. Definition: Given $\mathrm{X} \times \mathrm{Y}$, R is binary relation **iff**

$R \subseteq X \times Y$. Inspired by the Python syntax, we set

$$x, \triangleq \emptyset, x, \ \triangleq \{\{\emptyset\}, \{\emptyset, x\}\} \tag{2.10}$$

$$x, y, \triangleq (x, y) \triangleq \{\{x\}, \{x, y\}\} \tag{2.11}$$

$$x, y, z, \triangleq ((x, y), z) \quad \text{(provided } z \in Z) \tag{2.12}$$

and so on. Such original definition of (x, y) is really formal but, on the other hand, does not follow from the concept of mapping. It then avoids circular reasoning and ensures that $(x, y) = (x', y')$ **iff** $x' = x \wedge y = y'$. Remark that x, is the ordered pair $(0, x)$.

2.6 Functions and mappings

2.6.1 First definitions

Given a set X and a set Y, a function f is as a binary relation $G \subseteq X \times Y$ that satisfies:

$$((x, y) \in G \wedge (x, z) \in G) \Rightarrow y = z. \tag{2.13}$$

So, each function implicitely conveys such *graph* $G \subseteq X \times Y$, what we denote by $f : X \to Y$. Note that the degenerate case $f = \emptyset$ may occur. For instance $X = \emptyset$ forces $G = \emptyset$.
On the other hand, any pair (x, y) of G can be seen as a correspondence "from x to y" and is then written $x \mapsto y$. Such y may be written f(x) or y_x. It commonly referred as "the image of x by f". Bear in mind that $y_x = f(x)$ is necessarily unique, since the above definition says that $(x \mapsto y \wedge x \mapsto z) \Rightarrow y = z$. We say that f *maps* x *to* f(x) or, alternatively, that f *returns* f(x) *from* x. So, f(x) ranges over an *image* set f(X), as x ranges over the domain $\texttt{dom}(f) = \{x \in X : x \text{ is an actual input}\}$. Formally,

$$\text{dom}(f) \triangleq \pi_X(G) = \{x \in X : f(x) \text{ exists}\} \tag{2.14}$$

$$f(X) \triangleq \pi_Y(G) = \{y \in Y : y = f(x) \text{ for some } x\}; \tag{2.15}$$

where $\pi_X : X \times Y \to X, (x, y) \mapsto x$ and $\pi_Y : X \times Y \to Y, (x, y) \mapsto y$.
We say that f is a *mapping* **iff** $\texttt{dom}(f) = X$. The set of all mapping(s) $f : X \to Y$ is denoted by Y^X.

2.6.2 Inverse image

We say that y of Y has an *inverse image* x **iff** $y = f(x)$.

2.6.3 Ontoness

f is said to be *onto* **iff** $f(X) = Y$. In other words, each y of Y has an inverse image x.

2.6.4 Injectivity

f is said to be *one-to-one* (*injective*) **iff** each y of Y that has at most one inverse image x. In other words, each arrow $x \mapsto y_x$ is reversed to $y_x \mapsto x_y$; which defines $f^{-1} : f(X) \to X, y \mapsto x$ the *the inverse (mapping) of* f.
Remark that f^{-1} is a bijective) (see below) and that $(f^{-1})^{-1} = f$.

2.6.5 Bijectivity

f is said *bijective* (or, alternatively, *a bijection*) **iff** each y of Y has a unique inverse image x.
In other word, f is bijective **iff** f is both onto and one-to-one.

Remark that $\emptyset$ is bijective, since it is *vacuously true* that $S = \emptyset$ satifies every statemenent $(x, y) \in S \Rightarrow$ `conclusion`. To see that, assume, to reach a contradiction, that the statement is false as $S = \emptyset$, *i.e.* that there exists $(x, y) \in S = \emptyset$ such that `conclusion` fails. We have then reached a desired contradiction, namely, that $(x, y) \in S = \emptyset$. So ends the proof. Furthermore, each set X conveys a bijection $X \to X, x \mapsto x$; which is called the identity mapping (of X).

2.7 Composition and identification

2.7.1 Composition

Given $f : X \to Y$ and $g : Y \to Z$, we set

$$g \circ f(x) \triangleq g(f(x)) \tag{2.16}$$

where f(x) exists and is in `dom`(f). This defines a function $g \circ f : X \to Z, x \mapsto g(f(x))$ that is called the composition of f and g. Briefely, $x \mapsto y \mapsto z = g(y) = g(f(x))$. Note that $g \circ f$ is a mapping **iff** f is so and $f(X) \subset$ `dom`(g)
// TODO: Add remaks about systems and composing and OOP drawbacks

2.7.2 Composition of bijections

Given bijections $f : X \to Y, x \mapsto y$ and $g : Y \to Z, y \mapsto z$, the compositions $g \circ f$ and $f^{-1} \circ g^{-1}$ are bijective as well. To sum it up, the chain

$$\cdots \mapsto x \overset{f}{\mapsto} y \overset{g}{\mapsto} z \overset{g^{-1}}{\mapsto} y \overset{f^{-1}}{\mapsto} x \mapsto \cdots \tag{2.17}$$

does not break.

2.7.3 Identification

When there exists a bijection $b : X \to Y$, we may let us identify X with Y, and let $X \equiv Y$ denote such identification. The variant $X \equiv_b Y$ makes the bijection b explicit. Note that $X \equiv X$, and that $X \equiv Y$ combined with $Y \equiv Z$ implies $X \equiv Z$.

2.7.4 Identification and Cartesian product

$X \equiv A$, $Y \equiv B$, then $X \times Y \equiv A \times B$. The converse does not always hold!

2.8 Sequences

Finite sets

A set I if *finite* **iff** $I \equiv \{0, \ldots, n-1\} \subseteq \mathbf{N}$ for some $n \in \mathbf{N}$. Remark that the case $n = 0$ is the degenerate case $I = \{0, \ldots, \text{-}1\} = \emptyset$. Given a (finite) set $I \equiv \{0, \ldots, n-1\}$, n is univoquely valued, *i.e.*

$$\mathrm{I} \equiv_{\mathrm{b}} \{0, \dots, \mathrm{n}-1\} \equiv_{\mathrm{b}'} \{0, \dots, \mathrm{n}'-1\} \Rightarrow \mathrm{n} = \mathrm{n}'. \tag{2.18}$$

n is then an inner property of I that is called the cardinal of I and is denoted by $\mathtt{card}(\mathrm{I})$. Remark that $\mathtt{card}(\{0, \dots, \mathrm{n}-1\}) = \mathtt{card}(\mathrm{n}) = \mathrm{n}$, since $\mathrm{n} = \{0, \dots, \mathrm{n}-1\}$.

Infinite sets

A set is *infinite* **iff** it is not finite.

2.8.1 Countable sets

Countably infinite sets

A set I is *countably infinite* **iff** $\mathrm{I} \equiv \mathbf{N}$.

Countable sets

A set is *countable* **iff** either I is countably infinite or finite. For instance, sets $\mathbf{N}, \mathbf{Z}$, $\mathbf{Q}$ (the rational numbers) are countable.

Countable union of countable sets

Given finitely countably countable sets, the union of those sets is a countable set as well.

Uncountable sets

A set is *uncountable* **iff** it is not countable. For instance, the real line $\mathbf{R}$ is not countable. Since $\mathbf{R}$ countains the countable set $\mathbf{Q}$, there are uncountably many irrational (*i.e.* not rational) numbers. For instance, $\sqrt{2}$ is irrational.

2.8.2 Sequences: Definition

We say that a mapping $\mathrm{x} : \mathrm{I} \to \mathrm{X}$ is a sequence (over I, in X) **iff** I is countable.
If $\mathrm{I} = \emptyset$, then x is the empty sequence $[\ ] = \emptyset$.
More generally, if $\mathrm{I} = \{0, \dots, \mathrm{n}-1\}$, then x can then be seen as the list

$$(0, \mathrm{x}_0), \dots, (\mathrm{n}-1, \mathrm{x}_{\mathrm{n}-1}), \tag{2.19}$$

which justifies the more compact matrix notation $[\mathrm{x}_0 \ \cdots \ \mathrm{x}_{\mathrm{n}-1}]$, or $\mathrm{x}_0, \dots, \mathrm{x}_{\mathrm{n}-1}$ when no value x_i gets repeated.
The integer $\mathrm{n} = \mathtt{car}(\mathrm{I})$ is also known as the *length of the sequence.*
More generally, if $\{0, \dots, \mathrm{n}-1\} \equiv_{\mathrm{b}} \mathrm{I}$, then $[\mathrm{x}_\mathrm{i} \ \cdots \ \mathrm{x}_\mathrm{j}]$ denotes x as well; understood that $\mathrm{x}_\mathrm{i} = \mathrm{x}_{\mathrm{b}(0)}, \dots, \mathrm{x}_\mathrm{j} = \mathrm{x}_{\mathrm{b}(\mathrm{n}-1)}$
If I is a set of integers $\mathrm{i} = \mathrm{b}(0), \mathrm{i}' = \mathrm{b}(1), \dots$ such that $\mathrm{i} < \mathrm{i}' < \cdots$, then the two "versions" of $[\mathrm{x}_\mathrm{i} \ \mathrm{x}_{\mathrm{i}'} \ \cdots]$ (one is $\mathrm{i} \mapsto \mathrm{x}_\mathrm{i}, \mathrm{i}' \mapsto \mathrm{x}_{\mathrm{i}'}, \dots$, the other one is $0 \mapsto \mathrm{x}_\mathrm{i}, 1 \mapsto \mathrm{x}_{\mathrm{i}'}, \dots$), are identified each other. For instance, every sequence $1 \mapsto \mathrm{x}_1, \dots, \mathrm{n} \mapsto \mathrm{x}_\mathrm{n}$ shall be denoted by $[\mathrm{x}_1 \ \cdots \mathrm{x}_\mathrm{n}]$ and identified with the sequence $0 \mapsto \mathrm{x}_1, \dots, \mathrm{n}-1 \mapsto \mathrm{x}_\mathrm{n}$. Finite sequences are precisely the sequences over finite sets. Unsuprinsingly, sequences over countably infinite sets are... *infinite.* We will now see that finite sequences may be seen as a special case of infinite sequences.

2.8.3 Finite sequences: A Pythonic point of view

In Python, `list` and `tuple` are two standard implementations of finite sequences over $\{0, \dots n-1\}$. Given a finite sequence `x` of positive length n, `x[k]` means the value x_k $(0 \leq k < n)$. Moreover, calling `x[-1]`, ..., `x[-n]` returns `x[n-1]`, ..., `x[0]`. To justify this syntax, pick a sequence $x : \{0, \dots, n-1\} \to X$, then set

$$x_{\texttt{-1}} \triangleq x_{n-1}, \tag{2.20}$$

$$x_{\texttt{-2}} \triangleq x_{n-2}, \tag{2.21}$$

$$\vdots$$

so that $[x_0 \ \cdots \ x_{n-1}]$ is (identified with) any sequence

$$[x_1 \ \ x_2 \ \ \cdots \ \ x_{\mathbf{0}}], \tag{2.22}$$

$$[x_{\mathbf{0}} \ \ x_1 \ \ \cdots \ \ x_{\mathbf{n-1}}], \tag{2.23}$$

$$[x_{\texttt{\textbf{-1}}} \ x_0 \ \ \cdots \ \ x_{\mathbf{n-2}}], \tag{2.24}$$

$$[x_{\texttt{\textbf{-2}}} \ x_{\texttt{-1}} \ \cdots \ x_{n-3}], \tag{2.25}$$

$$\iddots$$

$$[x_{\texttt{-}\mathbf{n}} \ x_{\texttt{-}n+1} \ \cdots \ x_1], \tag{2.26}$$

$$\iddots .$$

2.9 Data, integers and finite sequences

2.9.1 Special case: Integers and rational numbers

The Euclidian division (say modulo $N \geq 2$) furnishes a way to encode integers as finite sequences of data. For instance, where $N = 2$, $1 = 2^0$ may identified with $[1 \ 0 \ \cdots]$, $2 = 2^1$ with $[0 \ 1 \ \cdots]$, $3 = 2^0 + 2^1$ with $[1 \ 1 \ 0 \ \cdots]$, and so on. Any of those sequences is now seen as a memory that stores the numerical value of an integer. We let the reader do the same with the usual decimal numbering $(N = 10)$. Note that appending a value that keeps track of the sign extends such encoding process to signed integers, then to all rational numbers. This is actually what we do when we write $\texttt{-}1/3 = \texttt{-}0,33\dots:$ (1) Only six symbols are sufficient to encode the rational number -1/3 as a sequence, (2) those symbols are taken from a finite set of symbols, and (3) there is always a way to get them in a finite amount of time (computation). In contrast, numerical values of irrational numbers cannot be encoded within a finite amount of memory. Forunately, (countably) infinitely many irrationals support finite definitions. For example, `sqrt`(2) is the only positive real number such that `sqrt`(2) $*$ `sqrt`(2) $==$ 2,

2.9.2 Memory as sequence of data

More generally, finite sequences provide an abstraction for what a memory is. A memory is (identified with) a finite sequence $[x_0 \ \cdots \ x_{n-1}]$; where x_0 is the first chunk of data, x_1 the second one (if $n > 1$), and so on. At a lower level, every chunk of data is encoded as a finite sequence of symbols, *e.g.* *0*, *1*, ..., *9*, *A*, *B*, ..., *Z*. Those symbols S exist in finite amount, so that a convention that encodes every S as an integer can be set. Computer theory assumes that such memory is potentially infinite, *i.e.* not bounded, in the sense that a memory x has positive length n and that n can always be increased on demand.

Chapter 3

Cartesian product (extended version)

3.1 Definition

Consider set(s) X, whose union is C: Choose a mapping $\mathrm{I} \to \mathrm{S}, \mathrm{i} \mapsto \mathrm{X_i}$ then define its *Cartesian product* as follows,

$$\prod_{\mathrm{i}\in\mathrm{I}} \mathrm{X_i} \triangleq \{\mathrm{f} \in \mathrm{S^I} : \mathrm{f(i)} \in \mathrm{X_i}\}. \tag{3.1}$$

Once the degenerate case $\mathrm{I} = \emptyset$ is put aside, any $\mathrm{f} : \mathrm{I} \mapsto \mathrm{X_i}, \mathrm{i} \mapsto \mathrm{x_i} \in \mathrm{X_i}$ is (definition) a *choice function*: The choice function f takes a set $\mathrm{X_i}$ then returns a "chosen" element $\mathrm{x_i}$ of $\mathrm{X_i}$.

In order to avoid conflict with our previous definition of the Cartesian product $\mathrm{X} \times \mathrm{Y}$, we keep assigning the first Cartesian product the symbol $\times$, as our new version is exclusively assigned the symbols $\prod$ and, temporarily, $*$.

3.2 Axiom of choice and Python's pop method

In ZFC, the axiom of choice asserts that $\mathrm{I} = \emptyset$ combined with $\mathrm{X_i} \neq \emptyset$ (for every i of I) implies that the Cartesian product $\prod_{\mathrm{i}\in\mathrm{I}} \mathrm{X_i}$ is nonempty. Equivalently, each $\mathrm{X_i}$ conveys (at least) one choice function $\mathrm{f} : \mathrm{i} \mapsto \mathrm{X_i}$. In Python, the `pop` method makes such choice, in the sense that `X.pop()` returns a random element `x` of the `set X`.

3.3 Finite Cartesian product

From now on, we suppose without loss of generality that $\mathrm{I} = \{0, \ldots, \mathrm{j}, \ldots, \mathrm{m}, \ldots, \mathrm{n}\}$ (where *j*, *m*, and *n* run through $\mathbf{N}$) and that no $\mathrm{X_i}$ is the empty set: The axiom of choice asserts that $\prod_{\mathrm{i}\in\{\mathrm{j}\ldots,\mathrm{m}\}} \mathrm{X_i} = \emptyset$ forces $\{\mathrm{j}, \ldots, \mathrm{m}\} = \emptyset$. Conversely, the "worst case" $\{\mathrm{j}, \ldots, \mathrm{m}\} = \emptyset$ means $\prod_{\mathrm{i}\in\{\mathrm{j}\ldots,\mathrm{m}\}} \mathrm{X_i} = \{\emptyset\}$. Any Cartesian product $\prod_{\mathrm{i}\in\{\mathrm{j}\ldots,\mathrm{m}\}} \mathrm{X_i}$ is usually denoted by $\prod_{\mathrm{i}=\mathrm{j}}^{\mathrm{m}} \mathrm{X_i}$. It will be temporarily be denoted by $[\mathrm{X_j} * \cdots * \mathrm{X_m}]$ as well. Clearly, $[\mathrm{X_j}] \equiv \mathrm{X_j}$, and $\prod_{\mathrm{i}=\mathrm{j}}^{\mathrm{m}} \mathrm{X_i} = [\mathrm{X_j} * \cdots * \mathrm{X_m}] = \{\emptyset\}$ whether $\{\mathrm{j}, \ldots, \mathrm{m}\} = \emptyset$.

3.4 The write and erase operations

Pick n in $\mathbf{N}$ then define an append operation, denoted by $*$, as follows

$$\begin{aligned} * : [\mathrm{X}_0 * \cdots * \mathrm{X}_{n-1}] \times \mathrm{X}_n &\to \prod_{i=0}^{n} \mathrm{X}_i \\ [\mathrm{x}_0 \cdots \mathrm{x}_n - 1], \mathrm{x}_n, &\mapsto [\mathrm{x}_0 \cdots \mathrm{x}_{n-1}\mathrm{x}_n] \end{aligned} \tag{3.2}$$

We can now define a write operation (mapping) over the union of all sets $[\mathrm{X}_0 * \cdots * \mathrm{X}_{n-1}]$ by setting

$$\mathtt{write}(\emptyset) \triangleq \emptyset = [\ ] \tag{3.3}$$

$$\mathtt{write}(\mathrm{x}_0,) \triangleq *(\mathtt{write}(\emptyset), \mathrm{x}_0,) = [\mathrm{x}_0] \tag{3.4}$$

$$\mathtt{write}(\mathrm{x}_0, \mathrm{x}_1,) \triangleq *(\mathtt{write}(\mathrm{x}_0,), \mathrm{x}_1,) = [\mathrm{x}_0\ \mathrm{x}_1] \tag{3.5}$$

$$\mathtt{write}(\mathrm{x}_0, \mathrm{x}_1, \mathrm{x}_2,) \triangleq *(\mathtt{write}(\mathrm{x}_0, \mathrm{x}_1,)\ \mathrm{x}_2,) = [\mathrm{x}_0\,\mathrm{x}_1\,\mathrm{x}_2] \tag{3.6}$$

and so on. Clearly, every write is onto. Furthermore, write has an inverse erase, as follows,

$$\mathtt{erase}([\mathrm{x}_0\ \cdots\ \mathrm{x}_n] \triangleq \mathrm{x}_0, \mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n,\ . \tag{3.7}$$

A possible point of view is that X_i now stores the deleted element x_i. We have thus established that

$$(((\mathrm{X}_0 \times \mathrm{X}_1) \times \cdots) \times \mathrm{X}_n \equiv \mathrm{X}_0 * \cdots * \mathrm{X}_n \tag{3.8}$$

The case $\mathtt{write}(\mathrm{x}_0,)$ is the case $\mathrm{X}_0 \equiv [\mathrm{X}_0] \triangleq \mathrm{X}_0^{\{0\}}$. The case $\mathtt{write}(\mathrm{x}_0, \mathrm{x}_1,)$ clearly shows that $\mathrm{X}_0 \times \mathrm{X}_1 \equiv \mathrm{X}_0 * \mathrm{X}_1$.

3.5 Concat and split operations

Provided the above *j*, *m*, *n*, we easily check that the following mappings concat and split

$$\begin{aligned} \mathtt{concat} : [\mathrm{X}_0 * \cdots * \mathrm{X}_m] \times [\mathrm{X}_{m+1} * \cdots * \mathrm{X}_n] &\to [\mathrm{X}_0 * \cdots * \mathrm{X}_n] \\ [\mathrm{x}_0\ \cdots\ \mathrm{x}_m], [\mathrm{x}_{m+1}\ \cdots\ \mathrm{x}_n], &\mapsto [\mathrm{x}_0\ \cdots\ \mathrm{x}_n] \end{aligned} \tag{3.9}$$

$$\begin{aligned} \mathtt{split} : [\mathrm{X}_0 * \cdots * \mathrm{X}_n] &\to [\mathrm{X}_0 * \cdots * \mathrm{X}_m] \times [\mathrm{X}_{m+1} * \cdots * \mathrm{X}_n] \\ [\mathrm{x}_0\ \cdots\ \mathrm{x}_n] &\mapsto [\mathrm{x}_0\ \cdots\ \mathrm{x}_m], [\mathrm{x}_{m+1}\ \cdots\ \mathrm{x}_n], \end{aligned} \tag{3.10}$$

are bijective and that split is the inverse of concat. Hence

$$(\mathrm{X}_0 * \cdots * \mathrm{X}_m) \times (\mathrm{X}_{m+1} * \cdots * \mathrm{X}_n) \equiv \mathrm{X}_0 * \cdots * \mathrm{X}_n \tag{3.11}$$

Moreover, we remark that the case $\mathrm{n} = 1$ is $[\mathrm{X}_0] \times [\mathrm{X}_1] \equiv \mathrm{X}_0 * \mathrm{X}_1 \equiv \mathrm{X}_0 \times \mathrm{X}_1$ (see above). It now makes sense to drop our initial definition of the Cartesian product $\mathrm{X}_0 \times \mathrm{X}_1$ in favor of

$$\mathrm{X}_j \times \cdots \times \mathrm{X}_m \triangleq \prod_{i=j}^{m} \mathrm{X}_i \quad (\mathrm{j} \leq \mathrm{m} \leq \mathrm{n}). \tag{3.12}$$

This second definition is an extension of the first one, but there is a little drawback. Indeed, the reader may have noticed that

$$\prod_{i\in\{0,1\}} X_i = \prod_{i=0}^{1} X_i = X_0 \times X_1 = X_1 \times X_0 = \prod_{i\in\{0,1\}} X_i. \tag{3.13}$$

Our new definition of the Cartesian product makes it then commutative. Therefore, every pair (x_0, x_1) of $X_0 \times X_1$ is no longer ordered, since $(x_0, x_1) = (x_1, x_0) \in X_1 \times X_0 = X_0 \times X_1$. Nevertheless, we will always stick to the lexicographic convention $X_0 \times \cdots \times X_j \times \cdots \times X_n$ $(0 \le j < n)$, so that $A \times B$ implicitely means $X_0 \times X_1$ (provided $X_0 = A, X_1 = B$), as $B \times A$ implicitely stands for $X_0 \times X_1$ where $X_0 = B, X_1 = A$. This assures that the notation (a, b) keeps expressive, in the sense that $(a, b) = (b, a)$ **iff** $a = b$. It is now clear that $B \times A \neq B \times A$ (unless $A = B$), but "losing" commutativity is not a real drawback. Actually, it brings clarity!

That being said, previous relations are then restated as

$$X_0 \equiv [X_0] \tag{3.14}$$
$$(((X_0 \times X_1) \times \cdots) \times X_n \equiv X_0 \times \cdots \times X_n \tag{3.15}$$
$$(X_0 \times \cdots \times X_m) \times (X_{m+1} \times \cdots \times X_n) \equiv X_0 \times \cdots \times X_n; \tag{3.16}$$

which is the associativity of the Cartesian product (to see that, take $n = 2$; $j = 0, 1.$). From now on, sequences $[x_0 \ \dots \ x_n]$ (matrix notation) can now be written alternatively $x_0, \dots, x_n$, (Python notation), or $(x_0, \dots, x_n)$ (tuple notation), or $(x_0, \dots, x_n)$ or $[x_0, \dots, x_n]$ (list notations).

Given a function $f : X_0 \times \cdots \times X_n \to Y$ we usually shorten $f(x_0, \dots, x_n,)$ to $f(x_0, \dots, x_n)$.

3.5.1 Commutativity of the extended Cartesian product

3.5.2 Cartesian power and closure under concatenation

The special case $X_i = X$ $(i = 0, \dots, n-1)$ is the Cartesian power

$$X^0 = X^{\emptyset} = \{\emptyset\} \tag{3.17}$$
$$X^1 = X^{\{0\}} \tag{3.18}$$
$$X^2 = X^{\{0,1\}} \tag{3.19}$$
$$\vdots$$
$$X^n = \prod_{i=0}^{n-1} X \qquad (n = 0, 1, 2, \dots). \tag{3.20}$$

Back to the definition of $*$ and $[X_n]$, we remark that the identification $X_n \equiv [X_n]$ turns `write` into a special case of `concat`: From now on, $*$ will now stand for `concat` as well.

The definition of $* =$ `concat` can be recursively extended, as follows

(3.21)
$$*(w_0, \dots, w_{n-2}, w_{n-1}) \triangleq *(*(w_0, \dots, w_{n-2}), w_{n-1}) \quad (w_0 \in X^{N_0} \dots, w_{n-1} \in X^{N_{n-1}}; n \ge 2);$$

provided natural number(s) $N_0, \dots, N_{n-1}$. It is easily shown by induction that such generalized $*$ still ranges over the whole union $X^0 \cup X^1 \cup X^2 \cup \cdots$.
Conversely, the inverse `split` of our new `*` is recursively defined as well. It is now clear that w is a finite sequence in X **iff** it is the concatenation of shorter sequences of X -what a breakthrough ;) In other words, X^* is the closure of X under $*$. So,

$$X^* = \bigcup_{n=0}^{\infty} X^n. \tag{3.22}$$

3.6 Indexing sequences from 0 or 1? Europe vs the US

In the US ground numbering starts with 1; which means that floors have numbers $2, 3, \dots$. In Europe, floors have numbers $1, 2, \dots$ and the street ground number 0. C programmers may find the European convention more consistent: The building being identified with a sequence x of data, the street floor being the first chunk x[0] of data, the first floor the second one x[1] and so on ...the sequence x is then ...the address of the building! Most people (including most programmers!) do not think of n as the set $\{0, \dots, n-1\}$ and do not write assertions like $0 \in 1$ (even it is perfectly fine to do so) but it may explain why it is fair to index from 0 instead of 1: $X^0 = X^\emptyset$, since $0 = \emptyset$, $X^1 = X^{\{0\}}$, since $1 = \{\emptyset\}$, $X^2 = X^{\{0,1\}}$, since $2 = \{0, 1\} = \{\emptyset, \{\emptyset\}\}$, and so on.

Chapter 4

Words and alphabets (TODO)

4.1 First definitions

By *alphabet* we mean any nonempty finite set Σ whose element(s) - the *character(s)* - will be combined into *words*. A word is then defined as a finite sequence of characters. Since there is no prior *grammatical* restriction to what such sequence must be, any finite sequence of characers is a word. So, “word” is a synomnym of “finite sequence” in the current context. Still **in the current context**, the *empty word* ε will now be an alternative name for $\emptyset$.

4.1.1 head, tail , prefix, suffix

Given a nonempty finite sequence $(w_0, \ldots, w_{n-1})$ $(n > 0)$, we define length (`len`), head (`head`), and tail (`tail`) as follows,

$$\mathrm{len}(x) \triangleq n \tag{4.1}$$

$$\mathrm{head}(x) \triangleq w_0 \tag{4.2}$$

$$\mathrm{tail}(x) \triangleq w_{n-1} \tag{4.3}$$

We extend the definition of `len` by setting the length of the empty sequence as 0.

4.2 Python implementations

`list` and `tuple` are then general implemenations for words. When our alphabet is more specifically an abstraction for a writing system, *e.g.* the latin letters *a, b, c, ...*, the string type `str` is a more obvious choice. Note that Python does not provide any abstraction for the characters themselves (compare with C and the `typedef char`) As an alternative, Python identifies a character with a 1-length sequence: In Python `'a' == 'a'[0]` is `True`. Actually, this is exactly what we do from the step that turns `write` a special case of `concat`. In short, we will never deal with `write`/`erase` operations: Instead, we will concatenate/split sequences of any length.

4.3 Concatenation

Our alphabet Σ is equipped operation the concatenation $*$; see previous chapter. In Python, concatenation over `list` or `str` objects is denoted by the infix `+`. Given words u and v, the concatenation $u * v$ is then the word w such that

$$w_i = u_i \text{ IF } i \leq \texttt{len}(u) \text{ ELSE } w_{i-\texttt{len}(u)} \tag{4.4}$$

Note that $\mathrm{u} * \varepsilon = \varepsilon * \mathrm{u}$.
You may think of u as an operator that takes v as input, so that u(v) = uv. In Python, this is what `extend` does : `u.extend(v) = u+v`. Note that we compose from left to right, since English and mathematical formulas are written from left to right.

Prefix and suffix

Given a word w, we let *p*, *s* range over Σ^* and say that p is a *prefix* - or equivalently, that s is a *suffix*, **iff** $\mathrm{w} = \mathrm{p} * \mathrm{s}$. Remark that each word w is both its own prefix and suffix, since $\mathrm{w} = \varepsilon * \mathrm{w} = \mathrm{w} * \varepsilon$.

4.3.1 Python implementation

Alphabet

Our encoding table will be Unicode. Every character is identified with a codepoint. This is not sufficient to encompass all human writing systems, *e.g.* Chinese ideograms, but this is enough for Latin alphabet, cyrillic alphabet, and abjads (Arabic, Persian, Hebrew).

Words

The primary implementation of words is words are list of positive integers. Of course words can be strings as well. The correspondence between the two types is:

$$\texttt{my\_string} ='' \texttt{.join(chr(x) vforv x in my\_listof\_integers)} \tag{4.5}$$
$$\texttt{my\_list\_of\_integers} = [\texttt{ord(x) for x in my\_string}] \tag{4.6}$$

The empty word is '' (in `string`) or `[]` (in `list`), or `( )` (in `tuple`).
Remark that `set('') == set([]) == set()`.

Prefix and suffix

`w[: n] , w[n: ], w[:-n]`

Chapter 5

The parsing automaton (TODO)

Chapter 6

The packages' structure

Chapter 7

The Command Line Interface (CLI) TODO

Bibliography

[1] TLA Toolbox. https://github.com/tlaplus/tlaplus/releases. Accessed: June 23, 2023.

[2] Dehornoy, Patrick. *Mathématiques de l'informatique*. Dunod, 1999.

[3] Lamport, Leslie. *Industrial Use of TLA+*. https://lamport.azurewebsites.net/tla/industrial-use.html. Accessed: June 23, 2023.

[4] Lamport, Leslie. *The TLA+ Video Course*. https://lamport.azurewebsites.net/video/videos.html. Accessed: June 23, 2023.

[5] Lamport, Leslie. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[6] Schwartz, Laurent. *Analyse I*. Hermann, 1997.

[7] Wikipedia. *Finite element method*. https://en.wikipedia.org/wiki/Finite_element_method. Accessed: June 23, 2023.

[8] Wikipedia. *Laplace's demon*. https://en.wikipedia.org/wiki/Laplace%27s_demon. Accessed: June 23, 2023.