MarkdownToLaTeX: Mathsheet

Gabriel Cordier

June 15, 2023

Contents

Chapter 1

What specification is (biographical note)

To shortly describe what specification is, I will start with the Leslie Lamport's definition[3]:

"A specification describes a universe in which things behave correctly."

For instance, 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. We will not aim at generality but we will nevertheless need some basic knowledge about differential equations.

For convenience we declare that $M_E > 0$ is the Earth's mass, as $M_S > 0$ is the Sun's mass. Our first level of abstraction consists in identifying E and S with their respective center of mass. We now make the customary (yet redundant) assumption that the Earth's orbit lies in the complex plane, so that the Earth's position at time t is a complex number. The smooth mapping $p : \mathbf{R} \to \mathbf{C}, t \mapsto p(t)$ now maps time t to the Earth's position at time t. Finally, we assume without loss of generality that our current time is $t = 0$, so that $p(0) = R > 0$; otherwise we would not be here to talk about the Earth!

Combined with Newton's gravitation law (which provides the positive constant K), the second Newton's law leads to

$$\ddot{p} + KM_S \frac{p}{|p|^3} = 0. \tag{1.1}$$

So let's be humble: This is a crazy hard problem.

Hopefully, we empirically know that $|p(t)|$ does not vary that much among time, hence the reasonable assumption that $|p| = R$. Our abstraction now turns to be

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

where $C = \sqrt{KM_S/R^3}$. So,

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

We then reach a [class of] universes in which things behave properly. More specifically, the solution nests (exactly) four possible universes:

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

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

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

Note that our model does not decide in which universe we are living. A Cauchy condition about $\mathrm{p}(0)$ and $\dot{\mathrm{p}}(0)$ would divide our options by two, but our scope statement did not mention such stronger assumption ...

I started to get interested in formal methods since I believed that a mathematical way of thinking could help in software development: Not only by sometimes proving something but also by **setting clear boundaries to problems**.

Clémenceau said that "war is a too serious matter to be left to militaries", and I think he was right. Programming should not be left to "coders".

There is a psychological reality: "Coders" may not like formal methods because they believe they are sufficiently smart to not waste time with them. It is sad to observe how smart people may believe that Laplacian worlds[6] are effective. In my humble opinion, being smart is not relevant:

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 language matters as well:

- 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 opposite:

A programmer is someone who programs, *i.e.* someone whoe 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.

In my life, I have seen bad implementation and some of them were mine. I keep writing programs but 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* [4]. 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 finite memory that stores the necessary context. The state machine was first written in TLA+ then tested within the TLA+ Toolbox[1].

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 chapters 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: [5][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 Liza , Marge, Homer and, ...oh ya:

Maggie", 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 (functions, real numbers, and so on), and in 99.9% of time, this is the right thing to do.

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

$$0 \triangleq \emptyset \tag{2.1}$$

$$1 \triangleq \{0\} = \{\emptyset\} \tag{2.2}$$

$$2 \triangleq \{0,1\} = \{\emptyset, \{\emptyset\}\} \tag{2.3}$$

$$\vdots$$

$$n+1 \triangleq \{0,\dots,n\} \tag{2.4}$$

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.* $x < y$ is $x \leq y$ with the extra information that $x \neq y$. The binary relations (see Cartesian product) $\geq, >$ are $\leq, <$ read from right to left: $y \geq x$ is $x \leq y$, $y > x$ is $x < y$. The set of all natural numbers is denoted by $\mathbf{N}$. Addition $+$ over $\mathbf{N}$ is extended as follows,

$$n + 0 \triangleq n \tag{2.5}$$

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

$\mathbf{N}$ can be embarked in $\mathbf{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$, the set of all integers. $\mathbf{Z}$ is equipped with a (commutative) generalized addition $+$ that can be reversed through substraction $-$ (in the sense that $n - n = n + (-n) = 0$). 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

$$\mathbf{N} \triangleq \{0, 1, 2, \dots\} \tag{2.7}$$

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

(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 $X \times Y$ is primarily defined as the set of all ordered pair(s) (x, y) ($x \in x, y \in Y$). Definition: R is binary relation **iff** $R \subseteq A \times B$ Inspired by the Python syntax, we set

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

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

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

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 $(x, y) = (x', y') \iff x' = x \wedge y = y'$ Note 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, we define a function f as a subset G of $X \times Y$ that satisfies:

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

Thus, each function implicitely conveys such *graph* G, what we denote as $f : X \to Y$. 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 $x \overset{f}{\mapsto} y$, or simply $x \mapsto y$ when there is no ambiguity Such y may be written f(x) or y_x. 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 that f returns f(x) from x. So, f(x) ranges over a set f(X), as x runs through 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** such $\texttt{dom}(f) = X$.
The set of all mappings $f : X \to Y$ is denoted by Y^X.

2.6.2 Onto, one-to-one and bijective functions

Let $f : X \to Y$ be a function.

2.6.3 Antecedent

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

2.6.4 Ontoness

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

2.6.5 Injectivity

f is said to be one-to-one (injective) **iff** each y of Y that has at most an antecedent x.
In other words, the arrow $x \mapsto y$ is reversed to $y \mapsto x$. In such case, the mapping $f^{-1} : f(X) \to X, y \mapsto x$ exists and is called the inverse of f.

2.6.6 Bijectivity

f is said bijective (or f is a bijection) **iff** each y of Y has a unique antecedent 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 $\emptyset$ satifies every statement $(x, y) \in \emptyset$ TODO: explain more. 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 $\texttt{dom}(f)$. 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 $f(X) \subset \texttt{dom}(g)$
// 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

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

is valid (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 mark $X \equiv Y$ such identification. The variant $X \equiv_b Y$ makes the bijection b explicit. Note that $X \equiv X$ and that $X \equiv Y$ and $Y \equiv Z$ implies $X \equiv Z$.

2.7.4 Identification and Cartesian product

$X \equiv A$, $Y \equiv B$, then $X \times Y \equiv A \times B$.

2.8 Sequences

2.8.1 Countable sets

Infinitely countable sets

A set I is infinitely countable **iff** $I \equiv \mathbf{N}$.

Finite sets

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

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

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

Countable sets

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

Infinite sets

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

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 $x : I \to X$ is a sequence (over I, in X) **iff** I is countable.
In other words, a sequence is defined as a member of X^I **iff** I is countable.
If $I = \emptyset$, then x is the empty sequence $[\,] = \emptyset$.
More generally, if $I = \{0, \dots, n-1\}$, then x can then be seen as the list

$$(0, x_0), \dots, (n-1, x_{n-1}), \tag{2.19}$$

which justifies the more compact matrix notation $[x_0 \ \cdots \ x_{n-1}]$, or $x_0, \dots, x_{n-1}$ when no value x_i gets repeated.
The integer $n = \texttt{car}(I)$ is the length of the sequence as well.
More generally, if $\{0, \dots, n-1\} \equiv_b I$, then $[x_i \ \cdots \ x_j]$ denotes x as well; understood that $x_i = x_{b(0)}, \dots, x_j = x_{b(n-1)}$
If I is a set of integers $i = b(0), i' = b(1), \dots$ such that $i < i' < \cdots$, then the two "versions" of $[x_i \ x_{i'} \ \cdots]$ (one is $i \mapsto x_i, i' \mapsto x_{i'}, \dots$, the other one is $0 \mapsto x_i, 1 \mapsto x_{i'}, \dots$), are "superimposed" (identified each other). For instance, every sequence $1 \mapsto x_1, \dots, n \mapsto x_n$ shall be denoted by $[x_1 \ \cdots x_n]$ and identified with the sequence $0 \mapsto x_1, \dots, n-1 \mapsto x_n$.
Finite sequences are precisely the sequences over finite sets. Unsunprinsingly, sequences over countably infinite sets are ...infinite. What we now call 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

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

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

$$\vdots$$

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

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

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

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

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

$$\ddots$$

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

$$\ddots.$$

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 expressions like -1/3 = -0.33...: (1) Only 6 symbols are necessary to encode the irrational 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, *e.g.* $\sqrt{2}, \pi, e$, can never be encoded within a finite amount of memory. Hopefully, (infinitely countably) many irrationals support finite definitions! For instance, $\sqrt{2}^2 = 2, e = \sum_{n=0}^{\infty} \frac{1}{n!}$, $2\pi = \min\{x > 0 : e^{ix} = 1\}$.

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, ..., 2. 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 a set S of set(s) X. Let C be the union of all set(s). Choose a mapping $I \to S, i \mapsto X_i$ then define its Cartesian product as follows,

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

Any element $f : i \mapsto X_i, i \mapsto x_i \in X_i$ is called choice function: The choice function f takes a set X then returns a "chosen" element x of X. In order to avoid conflict with our previous definition of the Cartesian product, we assign the first Cartesian product the symbol $\times$, as our new version is assigned the symbol $*$.

3.2 Axiom of Choice and Python's pop method

In ZFC, the Axiom of Choice asserts that $I = \emptyset$ combined with $X_i \neq X_i$ (for every i of I) implies that such product is never empty, *i.e.* $\prod_{i \in I} X_i \neq \emptyset$.
Equivalently, $\prod_{i \in I} X_i$ is never empty, unless I or an X_i is $\emptyset$. Another equivalent statement is that each X has a choice function $f : i \mapsto X$. 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 $I = \{0, \dots, j \dots, m, \dots, n\}$ for some n of $\mathbf{N}$ and that no X_i is the empty set. The axiom of choice then asserts that $\prod_{i \in \{j \dots, m\}}$ is never $\emptyset$. In the "worst case" ($I = \emptyset$), $\prod_{i \in \{j \dots, m\}} = \{\emptyset\}$. $\prod_{i \in \{j \dots, m\}}$ is usually denoted by $\prod_{i=j}^{m} X_i$. It will be temporarily be denoted by $[X_j * \cdots * X_m]$ as well. Remark that $[X_j] \equiv X_j$ and that $\prod_{i=j}^{m} X_i = [X_j * \cdots * X_m] = \{\emptyset\}$ when $j < m$.

3.4 The write and erase operations

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

$$(3.2) \qquad \begin{aligned} * : [X_0 * \cdots * X_{n-1}] \times X_n &\to \prod_{i=0}^{n} X_i \\ [x_0 \cdots x_n - 1], x_n, &\mapsto [x_0 \cdots x_{n-1} x_n] \end{aligned}$$

We can now define a `write` operation (mapping) over the union of all sets $[X_0 * \cdots * X_{n-1}]$ by setting

$$(3.3) \qquad \texttt{write}(\emptyset) \triangleq \emptyset = [\ ]$$

$$(3.4) \qquad \texttt{write}(x_0,) \triangleq *(\texttt{write}(\emptyset), x_0,) = [x_0]$$

$$(3.5) \qquad \texttt{write}(x_0, x_1,) \triangleq *(\texttt{write}(x_0,), x_1,) = [x_0\ x_1]$$

$$(3.6) \qquad \texttt{write}(x_0, x_1, x_2,) \triangleq *(\texttt{write}(x_0, x_1,)\ x_2,) = [x_0\, x_1\, x_2]$$

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

$$(3.7) \qquad \texttt{erase}([x_0 \cdots x_n] \triangleq x_0, x_1, x_2, \ldots, x_n, .$$

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

$$(3.8) \qquad (((X_0 \times X_1) \times \cdots) \times X_n \equiv X_0 * \cdots * X_n$$

The case $\texttt{write}(x_0,)$ is the case $X_0 \equiv [X_0] \triangleq X_0^{\{0\}}$. The case $\texttt{write}(x_0, x_1,)$ clearly shows that $X_0 \times X_1 \equiv X_0 * X_1$.

3.5 Concat and split operations

We easily check that the following mappings `concat` and `split`

$$(3.9) \qquad \begin{aligned} \texttt{concat} : [X_0 * \cdots * X_m] \times [X_{m+1} * \cdots * X_n] &\to [X_0 * \cdots * X_n] \\ ([x_0\ \cdots\ x_m], [x_{m+1}\ \cdots\ x_n]) &\mapsto [x_0\ \cdots\ x_n] \end{aligned}$$

$$(3.10) \qquad \begin{aligned} \texttt{split} : [X_0 * \cdots * X_n] &\to [X_0 * \cdots * X_m] \times [X_{m+1} * \cdots * X_n] \\ [x_0\ \cdots\ x_n] &\mapsto ([x_0\ \cdots\ x_m], [x_{m+1}\ \cdots\ x_n]) \end{aligned}$$

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

$$(3.11) \qquad (X_0 * \cdots * X_m) \times (X_{m+1} * \cdots * X_n) \equiv X_0 * \cdots * X_n$$

Moreover, we remark that the case $n = 1$ is $[X_0] \times [X_1] \equiv X_0 * X_1 \equiv X_0 \times X_1$ (see above). It now makes sense to drop our initial definition of the Cartesian product $X_0 \times X_1$ in favor of

$$(3.12) \qquad X_j \times \cdots \times X_m \triangleq \prod_{i=j}^{m} X_i \quad (j \leq m \leq n).$$

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. Nevertheless, we will always stick to the lexicographic convention $X_0 \times \cdots \times X_j \times \cdots \times X_n$ $(0 \leq j < n)$, 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 $[x_0 \ \ldots \ x_n]$ (matrix notation) can now be written alternatively $x_0, \ldots, x_n,$ (Python notation), or $(x_0, \ldots, x_n)$ (tuple notation), or $(x_0, \ldots, x_n)$ or $[x_0, \ldots, x_n]$ (list notations).

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

3.5.1 Commutativity of the extended Cartesian product

3.5.2 Cartesian power and closure under concatenation

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

$$X^0 = X^{\emptyset} = \{\emptyset\} \tag{3.17}$$

$$X^1 = X^{\{0\}} \tag{3.18}$$

$$X^2 = X^{\{0,1\}} \tag{3.19}$$

$$\vdots$$

$$X^n = \prod_{i=0}^{n-1} X \tag{3.20}$$

Back to the definition of $*$ and $[X_n]$, we remark that the identification $X_n \equiv [X_n]$ turns `write` as 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

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

provided natural number(s) $N_0, \ldots, N_{n-1},$. It is easily shown by induction such generalized $*$ still ranges over the whole union $\bigcup_{n=0}^{\infty} X^n$.

Conversely, the inverse `split` of our new $*$ is defined the same way. In other words, w is a finite sequence in X **iff** it is the concatenation of shorter sequences of X. What a breakthrough ;) We say that X^* is the closure of X under $*$. Thus,

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

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 $2, 3, \ldots$. In Europe, floors have numbers $1, 2, \ldots$ 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 $\{0, \ldots, n-1\}$ 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)

By *alphabet* we mean any nonempty finite set whose elements - understoods as the *characters* - will be combined into *words*. A word is then defined as a finite sequence of characters. Since there is no prior linguistic 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.

4.1 Length

Given a finite sequence $(y_1, \ldots, y_n)$, n is the length len(y) of y.

4.1.1 head, tail , prefix, suffix

Given a nonempty finite sequence $(y_1, \ldots, y_n)$ $(n > 0)$, we define length (len), head (head), and tail (tail) as follows,

$$\text{len}(y) \triangleq n \tag{4.1}$$

$$\text{head}(y) \triangleq y_1 \tag{4.2}$$

$$\text{tail}(y) \triangleq y_n \tag{4.3}$$

We extend the definition of len by setting the length of the empty sequence () as 0.
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** $w = p \cdot s$. Remark that each word w is both its own prefix and suffix, since $w = \varepsilon \cdot w = w \cdot \varepsilon$.

4.2 Concatenation

Our alphabet Σ is equipped operation the concatenation $\cdot$. Given words u and v, the concatenation $u \cdot v$ is the word w such that $w_i = u_i$ IF $i \leq \lambda(u)$ ELSE $w_{i-\lambda(u)}$. You may think of u as an operator that takes v as input, so that $u(v) = uv$. In Python, this is what extends does : u.extend(v) = uv. Note that we compose from left to right, since English and mathematical formulas are written from left to right. In such context $() = \emptyset$ is the empty word, denoted by ε.

4.2.1 Neutral element

Note that $u\varepsilon = \varepsilon u$

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** $w = p \cdot s$. Remark that each word w is both its own prefix and suffix, since $w = \varepsilon \cdot w = w \cdot \varepsilon$.

4.2.2 Head and tail

Given a nonempty word w, we define *head* and *tail* as the first (last) character of w, *i.e.* ‘head‘(w) = w_1, ‘tail‘(w) = $w_{\lambda(w)}$.

4.2.3 Implementation

Alphabet

Our encoding table will be Unicode. Every character is identified with a codepoint. Our alphabet is then $7, \ldots N$.

Words

The primary implementation of words is words are list of positive integers. Of course words can be strings The correspondence between the two types is: ‘my_string = ”.join(chr(x) for x in my_listof_integers)‘ ‘my_list_of_integers = [ord(x) for x in my_string]‘ Empty word is ‘”‘ (strings) or ‘[]‘. Remark that ‘set(”) == set([]) == set()‘. The concatenation is ‘u+v‘. ‘%s%s%(u, v).

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 15, 2023.

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

[3] Lamport, Leslie. *The TLA+ Video Course*. https://lamport.azurewebsites.net/video/videos.html. Accessed: June 15, 2023.

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

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

[6] Wikipedia. *Laplace's demon*, note = Accessed: June 15, 2023. https://en.wikipedia.org/wiki/Laplace%27s_demon.