



ft\_kleene

Language Theory and Automata

*Summary: how to automatically compute, encode, parse, and interpret information*

# Chapter I

## Foreword

The most important concept of the  $XX^{th}$  century is perhaps that of "computation". Those of you who recall the introduction on the history of mathematics (found in the module on set theory and boolean logic) might remember that it was the work surrounding Gödel's incompleteness theorems that paved the way for the birth of computer science. In a nutshell: once we knew the limits of what we *could* know, the question became how to better reach what we *can* know. In other words, the science of *logic* became the science of *computation*.

But since we had left off there, a big piece of the puzzle was missing, when it comes to "how computers were born". That missing piece is the mathematics of how we, humans, *express* computation. In practice, the biggest part of this edifice is language theory. Now, while language theory does have important links to linguistics (most importantly by providing mathematical grounding to the concepts of grammar, syntax, and semantics), it is first and foremost a *mathematical* theory: its subject is the study of processes (where a "process" is seen as strings of operations) and the symbolic expression (as well as generation or interpretation) of these processes.

So how do we situate language theory within the rest of computer science ? Well, it can be useful to refer to what we consider the main "models" of computation. To this day, there are 5 standard, frequently used classes of models used to express the idea of "computation". The classification that follows is adapted from [Raúl Rojas, \*Neural Networks, A Systematic Introduction\*, section 1.1.2](#). These are all equivalent, in the sense that within these models, there are constructs which are "universal" (aka "Turing-complete", a notion that we'll revisit). A model being "universal" basically means that "to our knowledge, anything that CAN be computed, given the limits of mathematics, can be computed by such a model". More simply put: a universal model is a theoretically well-founded model of computation as strong as we've ever achieved. (NB: Technical side note. The reason we say "to our knowledge", is because every "hyper-Turing" model of computation we've tried to invent was actually something we could simulate with another regular, "simple", Turing-complete model. This means that our attempt was at most Turing-complete itself, not "hyper-Turing". However, it was never *formally* proven that Turing completeness is the actual "speed limit" or "upper bound" of computation that is possible, so the question is *technically* still open.)

These 5 standard classes of models are:

- the mathematical model, which includes both language theory and the lambda calculus, and which emerged mostly from the works of Hilbert, Ackermann, Kleene and Church in the 1920s and 1930s, and saw major advances in the 1950s with the work of Chomsky and Kleene. It served to answer the question of what mathematical functions were computable, and under which conditions.
- the logic-operational model, which includes the original Turing Machine automaton (an imaginary computer invented by Turing in the 1930s), and which consists mostly of mathematically defining, on paper, a calculation apparatus capable, through mathematical logic, of computing functions.
- the physical computer model, which is mostly inherited from the original von Neumann architecture (ie, the abstract design which served for Z1 and ENIAC), and is where the quasi-totality of our modern computer engineering came from (excluding the current prototypes for quantum, biological, and neuromorphic computers).
- the cellular automaton model, also invented by von Neumann, and expanded by the likes of Conway and Wolfram, which surprisingly, can very much lead to [universality](#), as can be visualized in an awe-inspiring video at the end of this subject...
- the neural/biological model, which inspired the study of real and artificial neural networks from a computational standpoint; a work initiated by McCullochs, Pitts and Wiener in the 1940s, and which gained a lot of traction in the last decade with the advent of GPU/big data-based deep learning and machine learning. In this section, we could also add "close cousins" of neural networks, such as [Petri nets](#) and [bayesian networks](#), which represent dynamical or probabilistic systems via state transition graphs.
- Do note that there are also non-standard models of computation, which are less studied, less well understood, or simply less standard. E.g., it was shown that logic gates (and higher levels of computation) could be built over DNA and enzymes in bacteria using epigenetic tinkering; and forms of learning were exhibited in neuronless creatures such as slime molds. Quantum computation can also be included in this section, at this point in time.

These various fundamental models of computation do share a bunch of profound links, most of which we (sadly) won't be exploring in this module.

We will be concentrating on aspects of the first two members in this list: language theory and automata theory. It is through language theory that the code of programming languages (the one a human can read and write) can automatically be turned into binary code (the kind that a machine can execute). This means it's through language theory that these programming "languages" can be used to interpret mathematical ideas that a computer can then run. Automata provide a way of understanding languages and grammars via diagrams, and for this reason are generally taught during any language theory course. Of most interest, it is through these tools that one can build compilers; though language theory is also used in NLP, here and there.

## I.1 General Rules

- For this module, function prototypes are described in Rust, but you may use the language you want.
- 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 II

## Introduction

### II.1 Overview: vocabulary and the Chomsky hierarchy

The goal of this project is to help you reach a good literacy of language theory. What does that imply ? Well, in practice, at the end of this project, you should be able to pick up a textbook on "how to write your own compiler", and be competent enough to read and use it as a resource.

It should also help boost your Ctrl+F and "Find and Replace" IDE skills to a whole new realm of possibilities ! (You have NO idea how much time you will save when coding, honestly, it's game changing.)

There are a few key language theoretic concepts to discover before diving in. Don't worry if you don't understand everything immediately, this is just a way to help you start mapping ahead the essential parts of this new territory in your mind ! Come back to this section regularly.

- an **alphabet**  $A$  is a set of **symbols**, for example  $B = \{a, b\}$  is a representation of the binary alphabet.

- a **language**  $L_A$  is a set of "words" (also called "strings") over an alphabet  $A$ . Yes, these are the strings you know and love; just seen from the point of view of mathematics. For example,  $L_B = B^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$  is a language over the binary alphabet, called the "free language" over the binary alphabet. The "epsilon" symbol, written as  $\epsilon$  or  $\varepsilon$ , is used to describe the empty string:  $\epsilon = ""$ . The "star" operator in the expression  $B^*$  (called the *Kleene star*) is an important concept; one which we'll revisit. Another example of a language over the binary alphabet is the language  $L'_B = \{a^n b^n, n \in \mathbb{N}\} = \{\epsilon, ab, aabb, aaabbb, \dots\}$ . All languages over an alphabet are subsets of the free language over this alphabet.

If it makes sense, a language can be outfitted with the concatenation operator. For example, concatenation (often written as  $\cdot$  or  $+$ ) works in  $L_B$ , since any pair of strings

from  $L_B$  can be concatenated to return a string from  $L_B$ . This is not the case for  $L'_B$ , since, for example,  $ab + aabb = abaabb$  is not a valid string of  $L'_B$ . However, you can consider the language  $(L'_B)^*$  (called the Kleene star of  $L'_B$ , or the closure through concatenation of  $L'_B$ ), consisting of all concatenations of strings of  $L'_B$ , and in which  $abaabb$  is a valid string, but  $ba$  is not.

- a **grammar**  $G$  is a set of *production rules* over an alphabet  $A$  (set of letters, or *terminal symbols*; generally lowercase), combined with a set of variables  $V$  (*non-terminal symbols*; generally uppercase; note that "variable" is used in the mathematical sense of the term here, ie, a placeholder for a set of possible values, not a memory block containing a value). This triple pairing allows one to generate a complex language  $L_A$  over this alphabet. A grammar *produces* strings (of terminal symbols). A production rule is a way to transform a string containing non-terminal symbols into another string (containing possibly both terminal and non-terminal symbols). Symbols are said to be terminal if they cannot be transformed into other symbols via any production rule (ie, they are the "termination" of a chain of transformations), and non-terminal if they can (they can still be transformed into something else). Production rules define a way to rewrite the left-hand side of an  $A \rightarrow B$  pair into the right-hand side. For this reason, grammars are often also called *rewriting systems*.

Here is an example: we define our grammar  $G$  by taking the binary alphabet  $B = \{a, b\}$ , the set of variables  $V = \{S\}$  containing a single non-terminal  $S$ , and the single production rule  $S \rightarrow aSb/\epsilon$ . This rule means that any time you see the non-terminal  $S$ , it can be replaced either by the string  $aSb$  (which is a mix of terminals and non-terminals) or by the empty string  $\epsilon$ . The initial string to the grammar is " $S$ ". Try to see how the language generated by the grammar  $G$  is precisely the language  $L'_B$  seen above.  $G$  is an example of a *context-free grammar*.

NB: By convention, you can omit what the root input to a grammar is, and by this we mean that the rule containing  $S$  as its single input is always the first rule to be read; or you explicitly define the initial string (of terminals and/or non-terminals) from which the language can be inferred. These two ways of defining grammars are equivalent, since for any choice of initial string  $ABtCAD\dots$ , you can always just define a rule  $S \rightarrow ABtCAD\dots$  to be the start rule.

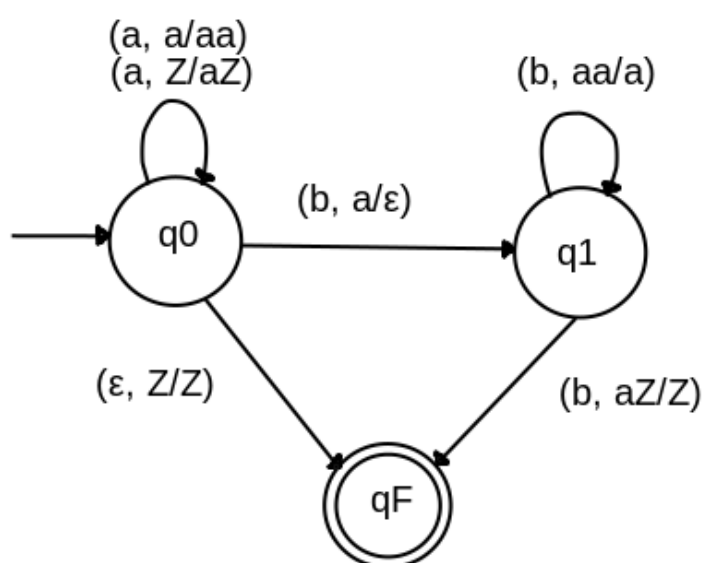
- an **automaton** is a mathematical model that allows the validation (execution) of a string of a given language. An automaton *consumes* strings (of terminal symbols). If you understand each terminal symbol as an atomic operation in a process, then an automaton executes this process step-by-step by reading each letter of an input string. Automata are generally represented through the use of *state transition diagrams*.

Here is the automaton for the grammar  $G$  that accepts the strings of  $L'_B$ . Note, it is a "pushdown automaton" (which is the kind of automaton specific to context-free grammars). In the diagram below:

- The circles are called the **states**; the arrows between the circles are called the state **transitions**: each time a letter of the input string is read, we go through a transition and reach another state.

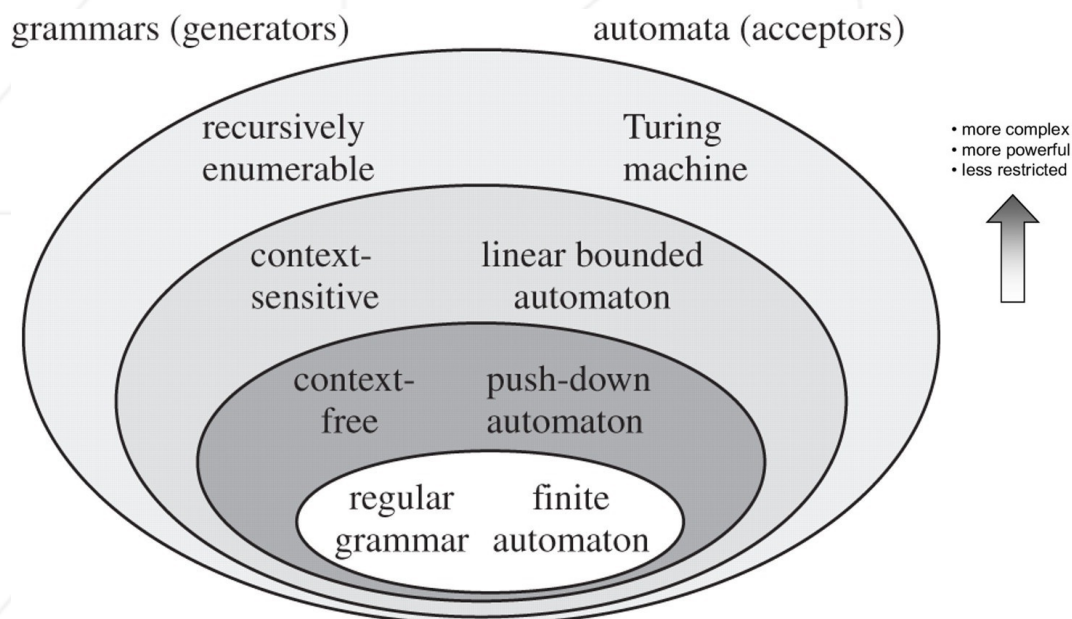
- The double-circle is called the "final state" or the "end state": if it can be reached, then your string is a valid string for the language described by this automaton.
- $Z$  represents the empty stack, and the  $(x, y/z)$  parts should be read as "if your next terminal symbol is  $x$ , and  $y$  is at the top of your stack, go through this arrow, and replace the top of your stack by  $z$ ". Note that if  $x$  is the empty string, one can always choose to go through that arrow (so long as the top of the stack matches  $y$  for the transition rule). Also, if  $y$  is  $\epsilon$ , no condition is given for the top of the stack.

Now, how does this automaton work ? You first start by arriving with any string in the state  $q_0$ . If you've started with the empty string, you can reach the end state immediately. If you have a string that starts with  $a$ , the automaton can read any number of  $a$ 's as it desires; for each  $a$  it reads, it pushes a new  $a$  to the stack each time: this will serve to keep count. If your string is non-empty and doesn't start with a series of  $a$ 's, the automaton fails. Once the  $a$ 's are done, since the stack has an  $a$  on top, you consume a  $b$  from your input string and pop an  $a$  from the stack to reach  $q_1$ . The number of  $b$ 's leftover in the empty string must match the number of  $a$ 's left in the stack so that  $q_F$  can finally be reached; it is reached by consuming the last  $a$  in the stack with the last  $b$  in the input string.



There is at least one automaton we can design for each grammar, and at least one way to formulate a grammar per automaton. There are 4 standard types of grammars/automata, organized in what is called the **Chomsky hierarchy**.





Language	Automaton	Grammar	Recognition
Recursively Enumerable Languages		Unrestricted $Baa \rightarrow A$	Undecidable 
Context-Sensitive Languages		Context Sensitive $A t \rightarrow aA$	Exponential? 
Context-Free Languages		Context Free $S \rightarrow gSc$	Polynomial 
Regular Languages		Regular $A \rightarrow cA$	Linear 

Note that *regular grammars* are called Type-3 grammars, *context-free grammars* called



Type-2, *context-sensitive grammars* called Type-1, and *recursively enumerable grammars* called Type-0. Any Type- $n$  grammar is also a Type- $m$  grammar if  $n \leq m$ , but the converse is not true. This means, for example, that a pushdown automaton (Type-2 level) can recognize a regular language (Type-3 level) as valid, but a finite automaton (Type-3 level) cannot recognize a language that is context-free but not regular (Type-2 level).

Be careful as to what someone means when they say a programming language "*is* Type- $n$ ". When describing a programming language, one should always distinguish the level of the Chomsky hierarchy required to parse/interpret/compile the language (which is generally Type-2 or, more frequently, Type-1) and the level of expressivity of the language (which is almost always Type-0). Some confusion may arise for this reason in the documentation you'll find, so be careful. Once you keep this distinction in mind, things are generally clear from context.

Throughout the course of this module, we will go up the Chomsky hierarchy, starting from the simplest, least expressive languages (those generated by regular grammars).

In all exercises, **parsing strings will be done from left to right**. We chose this because it is more intuitive from a programming standpoint, unlike the mathematical convention which tends to parse strings from right to left.

Further reading:

- Great general explanation of language theory concerning the tools relevant to compiler design (Regex, lexers, parsers, and), sadly there's a few errors in the examples: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_quick\\_guide.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_quick_guide.htm)
- Good lexicon, good images; a couple of unclear infos: <https://devopedia.org/chomsky-hierarchy>
- A website with good UX to learn and practice the syntax used by most modern Regex engines: <https://regexr.com/>
- A long text on the performance aspects of Regex: <https://swtch.com/rsc/regex-p/regexpl.html>
- More advanced examples of computational models: <https://cs.stackexchange.com/questions/140841/ram-or-von-neumann-which-theory-model-is-the-consensus>
- Something closer to a standard language theory course, with sadly a couple of mistakes in the examples: <https://scanftree.com/automata/>
- Some good articles in this course by Cornell university: <https://www.cs.cornell.edu/courses/cs2800/>
- An exotic use of grammars for the curious: <https://cpb-us-w2.wpmucdn.com/sites.coecis.cornell.edu/files/2016/01/plg0rx.pdf>
- a phenomenal, in-depth caricature of the various subjects in this text, by the youtuber Junferno: <https://www.youtube.com/watch?v=nVFvMboFFNc>

# Contents

<b>I</b>	<b>Foreword</b>	<b>1</b>
I.1	General Rules . . . . .	3
<b>II</b>	<b>Introduction</b>	<b>4</b>
II.1	Overview: vocabulary and the Chomsky hierarchy . . . . .	4
<b>III</b>	<b>Exercise 00 - Concat</b>	<b>11</b>
<b>IV</b>	<b>Exercise 01 - Power concat</b>	<b>12</b>
<b>V</b>	<b>Exercise 02 - Kleene star generator</b>	<b>13</b>
<b>VI</b>	<b>Exercise 03 - Levenshtein distance</b>	<b>15</b>
<b>VII</b>	<b>Interlude - Decision problems</b>	<b>16</b>
VII.1	Entscheidungsproblem . . . . .	16
VII.2	The Halting problem . . . . .	16
<b>VIII</b>	<b>Exercise 04 - Kleene star acceptor</b>	<b>18</b>
<b>IX</b>	<b>Interlude - Type-3 constructs of the Chomsky hierarchy</b>	<b>20</b>
IX.1	Regular languages, regular expressions, regular grammars and finite automata . . . . .	20
<b>X</b>	<b>Exercise 05 - Example regular language acceptor</b>	<b>24</b>
<b>XI</b>	<b>Exercise 06 - Finite automaton</b>	<b>26</b>
<b>XII</b>	<b>Exercise 07 - Deterministic finite automaton</b>	<b>28</b>
<b>XIII</b>	<b>Exercise 08 - Deterministic finite automaton minimization</b>	<b>29</b>
<b>XIV</b>	<b>Exercise 09 - Regex matching</b>	<b>31</b>
<b>XV</b>	<b>Interlude - Higher-level constructs of the Chomsky hierarchy</b>	<b>33</b>
XV.1	Limitations of finite automata . . . . .	33
XV.2	Type-2: Context-free grammars and pushdown automata . . . . .	33
XV.3	Type-1: Context-sensitive grammars and linear-bounded automata . .	34
XV.4	Type-0: Recursively enumerable grammars and Turing machines . . .	35
<b>XVI</b>	<b>Exercise 10 - Example context-free language acceptor</b>	<b>37</b>
<b>XVII</b>	<b>Exercise 11 - Example context-sensitive language acceptor</b>	<b>39</b>

<b>XVIII Interlude - Kleene's operators and category theory</b>	<b>41</b>
XVIII.1 Reminders on category theory . . . . .	41
XVIII.2 Kleene's special operators seen as functors . . . . .	42
XVIII.3 Going further: the free group functor . . . . .	42
<b>XIX Exercise 12 - Bonus: An intermediate-level RegEx engine</b>	<b>44</b>
<b>XX Interlude - another important model of computation</b>	<b>46</b>
XX.1 The $\lambda$ -calculus . . . . .	46
XX.2 The Church-Turing thesis . . . . .	48
<b>XXI Exercise 13 - Bonus: A small <math>\lambda</math>-calculus compiler</b>	<b>49</b>
<b>XXII Interlude - A final digression: universal cellular automata</b>	<b>52</b>
XXII.1 Conway's Game of Life . . . . .	52
XXII.2 Acknowledgements . . . . .	54

# Chapter III

## Exercise 00 - Concat

<b>Λ</b>	Exercise : 00
Concat	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(\ s_1\  + \ s_2\ )$	
Maximum space complexity : $O(\ s_1\  + \ s_2\ )$	

### III.0.1 Goal

The goal is to write a function which concatenates two strings together (yes, we are starting smoothly).

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

Let  $\Sigma$  be an arbitrary alphabet and  $s_1$  and  $s_2$  be arbitrary words over this alphabet.

The function must return the word  $s_1s_2$ .

The prototype of the function to write is the following:

```
fn concat(s1: &str, s2: &str) -> String;
```

# Chapter IV

## Exercise 01 - Power concat

<b>Δ</b>	Exercise : 01
Power concat	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(\ s\  \times m)$	
Maximum space complexity : $O(\ s\  \times m)$	

### IV.0.1 Goal

The goal is to write a function which concatenates the same word several times.

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

Let  $\Sigma$  be an arbitrary alphabet and  $s$  be an arbitrary word over this alphabet.


The function must return the word  $s^m$ .

The prototype of the function to write is the following:

```
fn power_concat(s: &str, m: usize) -> String;
```

# Chapter V

## Exercise 02 - Kleene star generator

	Exercise : 02
	Kleene star generator
	Allowed mathematical functions : <b>None</b>
	Maximum time complexity : $O(\binom{2n}{n})$
	Maximum space complexity : $O(\binom{2n}{n})$

### V.0.1 Goal

The goal is to write a function which returns the Kleene star over a given alphabet, up to the  $m$ -th rank of combinations.

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

Let  $\Sigma$  be an alphabet and  $a_i$  be its  $n$  symbols.

The function must return an array of all words composed of up to  $m$  symbols (words of length  $m$  included) of  $\Sigma$ . Since this can quickly become *very* costly in both time and space (see [here](#) and [here](#)), the function is not expected to work for high combined values of  $m$  and  $n$ .

The prototype of the function to write is the following:

```
fn kleene_star(sigma: &str, m: usize) -> Vec<String>;
```





# Chapter VI

## Exercise 03 - Levenshtein distance

<b>Δ</b>	Exercise : 03
Levenshtein distance	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(\ s_1\  \times \ s_2\ )$	
Maximum space complexity : $O(\ s_1\  \times \ s_2\ )$	

### VI.0.1 Goal

The goal is to write a function which computes the Levenshtein distance (also called edit distance) between two given strings.

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

Let  $\Sigma$  be an alphabet and  $s_1$  and  $s_2$  be strings over this alphabet.

The function must return the Levenshtein distance between  $s_1$  and  $s_2$ .

The prototype of the function to write is the following:

```
fn levenstein_distance(s1: &str, s2: &str) -> usize;
```

# Chapter VII

## Interlude - Decision problems

### VII.1 Entscheidungsproblem

In 1928, mathematicians David Hilbert and Wilhelm Ackermann, convinced by the theory of positivism, asked one of the most famous problems in mathematics: the **Entscheidungsproblem** (from German: "decision problem").



For a reminder on the philosophical doctrine of positivism, refer to the introduction of the subject **Ready, Set, Boole!**

This problem asked whether there was a mechanical procedure (an algorithm) which is able to tell whether a given statement is universally valid.

To solve this problem, it was first required to pin down a definition of an algorithm. This was done by Alonzo Church in 1935 with his theory of  $\lambda$ -calculus and Alan Turing in 1936 with his concept of Turing machines.

They both independently proved the Entscheidungsproblem is undecidable. Church proved that there is no computable function which can decide whether two given  $\lambda$ -calculus expression are equivalent. Whereas Turing reduced the problem to the **Halting problem**.

### VII.2 The Halting problem

The Halting problem is one of the most important decision problem in computer science and mathematics.

It is considered to be one of the first instances in history of a mathematical problem

proven to be undecidable (if not the first). The proof was given by Alan Turing in 1937 in his paper, "On Computable Numbers With an Application to the Entscheidungsproblem" (which can easily be found online, but is not easy to grasp at all).

The problem is about determining whether, for a given computer program  $A$  and an input  $B$ , the program will necessarily end, or if it will run forever.

Let  $g$  be a **total computable function** (a function which doesn't have any undefined behaviour for any given input) which takes as input a function  $f$  and its input  $i$  and tells whether the function  $f$  will halt. This means that we postulate that this magical "oracle"  $g$  function can solve the Halting problem for all functions  $f$  with all possible inputs  $i$ .

Let  $f$  be a function and  $i$  an input to that function. Now that we have our "oracle" function  $g$ , let's define our specific function  $f$  as follows (in pseudocode):

```
fn f(i) {  
  // Test whether f will halt with input i  
  if g(f, i) {  
    while true {  
      // Infinite loop  
    }  
  }  
}
```

In case the function  $g$  returns true (meaning that  $f$  halts), then the function  $f$  will loop forever. Contradiction.

In case the function  $f$  is expected to loop forever, the function  $g$  will return false, and so the function  $f$  will return. Contradiction.

What makes this function special, like Gödel's proof of the first incompleteness theorem, is the self-reference of  $f$ , which uses both  $g$  and, importantly, *itself* in its definition.  $f$  is recursive in a very special way: it is recursive in a way that forces  $g$ , which is "always right", into a contradiction. This leads to a form of computational "Liar's paradox", so to speak. Postulating the existence of a function like  $g$  which could solve the Halting problem in all cases leads to a logical paradox, thus a function like  $g$  cannot exist. Conclusion: the Halting problem is undecidable for "all functions in general" (though it can be solved from some simple classes of functions with specific inputs, of course). You can't have a function  $g$  which can predict the behavior of all functions  $f$  in advance, just from reading their source code. Some functions, you can only understand and study their behavior if you actually run them.

# Chapter VIII

## Exercise 04 - Kleene star acceptor

<b>A</b>	Exercise : 04
	Kleene star acceptor
	Allowed mathematical functions : <b>None</b>
	Maximum time complexity : $O(\max(n, m))$ or $O((n + m) \times \log(m))$
	Maximum space complexity : $O(\max(n, m))$ or $O((n + m) \times \log(m))$

### VIII.0.1 Goal

The goal is to write a function which can recognize whether a given word is valid for the language generated by the Kleene star of a given alphabet.

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

### VIII.0.2 Instructions

Let  $\Sigma$  be an alphabet with  $\#\Sigma = m$  symbols. Let  $\Sigma^*$  be the language generated by the Kleene star over this alphabet.

The function must return whether a given word  $s$  (of length  $n$ ) is valid over  $\Sigma^*$ .

Concerning complexity: this function can run in linear time (to be precise,  $O(\max(n, m))$ ), for a reasonably sized alphabet  $\Sigma$ . This means that a "naive" function running in  $O(n \times m)$  is invalid. A function running in  $O((n + m) \times \log(m))$  is also valid, so long as the student can explain the idea "reasonably sized alphabet" and why failing to have this can cause problems with the  $O(\max(n, m))$  implementation. Hint: everything depends

on the data structure used to store and test your alphabet.

The prototype of the function to write is the following:

```
fn kleene_star_acceptor(sigma: &str, s: &str) -> bool;
```

# Chapter IX

## Interlude - Type-3 constructs of the Chomsky hierarchy

### IX.1 Regular languages, regular expressions, regular grammars and finite automata

The simplest, least computationally rich, layer of the Chomsky hierarchy is that which corresponds to **regular grammars**, also called *Type-3 grammars*. These were formalized in the 1950s by Stephen Cole Kleene. Regular grammars generate **regular languages**, and strings of these regular languages can be validated by what are called **finite automata** (for which there exist a kind of state transition diagram).

NB: An example of a language which is Type-2 but not Type-3 was given in the introduction above (ie, the grammar generating the language  $\{a^n b^n \mid n \in \mathbb{N}\}$ ). Here, we'll be looking at the simpler layer.

An alternative way to represent a regular language is a regular expression: a regular expression can be thought of as an "augmented string"; it is a pattern which can be used to match a string of characters, or not match it. In practice, regular expressions are an advanced use of the Ctrl+F "Find" feature used in the quasi-totality of software that use strings.

Do note, though, that by abuse of language, most advanced RegEx software available today (everything from 'sed' to 'perl' to 'awk') are actually Turing-complete (ie, they are not limited to Type-3 grammars, and can actually act as languages with Type-0 expressivity), since they can use multiple variables, generalized conditions, etc. "Pure" regular expressions, in the technical sense of the term, do not even have a concept of memory; the only thing they have is the state of "where we are at in the parsing of our input string, right now".

### IX.1.1 Regular grammars

Regular grammars are defined as being limited to production rules of the form:

$$\begin{aligned} A &\rightarrow x \\ A &\rightarrow Bx \end{aligned}$$

for a "left-regular" grammar, or:

$$\begin{aligned} A &\rightarrow xB \\ A &\rightarrow x \end{aligned}$$

for a "right-regular" grammar; where  $A, B \in V$  (i.e.  $A$  and  $B$  are non-terminals), and  $x \in T^*$  (i.e.  $x$  is a string consisting only of terminals, which includes the empty string). Note that a grammar with production rules of the form:

$$\begin{aligned} A &\rightarrow xB \\ A &\rightarrow x \\ A &\rightarrow Bx \end{aligned}$$

might not be regular.

Let us give the an example of a regular grammar over the binary alphabet;  $G_1 = (V, \Sigma, P, S)$

$$\begin{aligned} V &= \{S, A, E\} \\ \Sigma &= \{a, b\} \\ P &= \begin{cases} S \rightarrow Ab \\ A \rightarrow Aa \mid Ab \mid Ea \mid Eb \\ E \rightarrow a \end{cases} \end{aligned}$$

This grammar generates the set of strings over  $A = \{a, b\}$  which start with  $a$ , end with  $b$ , and have at least three letters in total. Note that even if it isn't expressed in the neat form above (where it is easily recognizable as left-regular grammar) a grammar might still be regular. Eg,  $G_2$  such that:

$$\begin{aligned} V &= \{S, A\} \\ \Sigma &= \{a, b\} \\ P &= \begin{cases} S \rightarrow aAb \\ A \rightarrow AA \mid a \mid b \end{cases} \end{aligned}$$

generates precisely the same regular language as above: ie, we have  $G_1 = G_2$ . Now, did you understand what kind of language this grammar creates ? Once you have an



idea, read the next section for the solution !

### IX.1.2 Regular languages and regular expressions

We saw that grammars generate languages (which, themselves, are simply sets of strings). As mentioned at the top of this section, a regular expression is an alternative way to describe a language, via a special kind of string with its own kind of operators within it.

For example, as a regular expression, the language  $L$  generated by the grammars  $G_1$  and  $G_2$  can be expressed as the string  $a(a|b)^+b$ , where the 'superscript plus' symbol is called the *Kleene plus*. It basically does the same thing as the Kleene star, but excludes the empty string from the resulting language. This syntax is precisely what you would use in a Ctrl+F textbox if you wanted to find "any string starting with  $a$ , ending with  $b$ , and with an arbitrary (but nonzero) amount of  $a$ 's and  $b$ 's, in any order, in the middle".

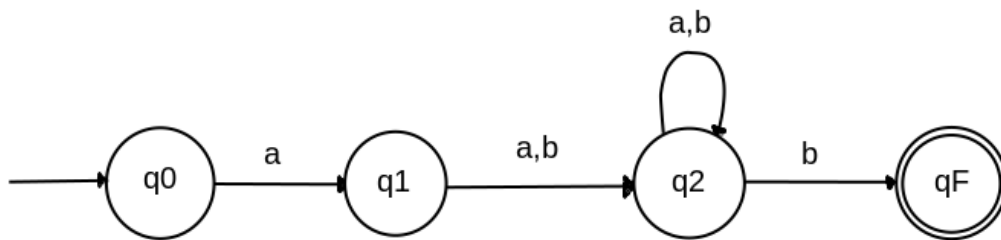
### IX.1.3 Finite automata; deterministic and non-deterministic

A decision problem is a kind of problem which has only two possible outputs: yes or no. For example: "Is  $x$  a prime number?" (where  $x$  is given as input) is a decision problem. Usually for a given decision problem, the goal is to find an algorithm which can answer the question for any given input. Telling whether a given string of characters  $s$  matches a given regex  $r$  is an example of a decision problem.

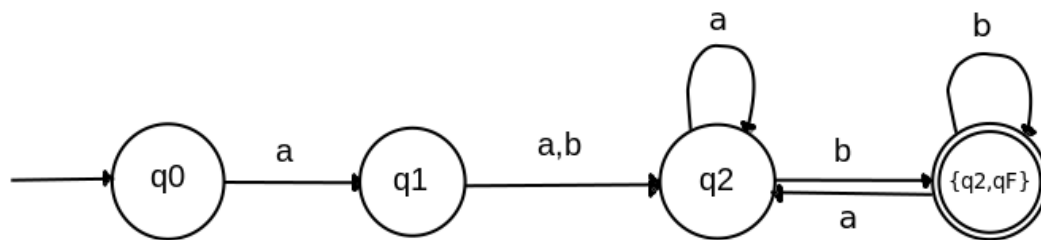
The different algorithms used to answer this problem in language theory are a central part of this module, and form the core of automata theory.

An automaton is a mathematical model of a machine which can accept or refuse a given string. They are thus a model for a specific decision problem. They are generally represented as state-transition diagrams (finite automata (FA) for Type-3 grammars, pushdown automata (PDA) for Type-2 grammars), or as "memory tape" diagrams (linear bounded automata (LBA) for Type-1 grammars, and Turing machines (TM) for Type-0 grammars). All of these automata come in possible two flavors: **deterministic** and **non-deterministic**. For a deterministic automaton, at every step in your sequence of reading a string, there is only a single next step that is possible. For a non-deterministic automaton, you might have a choice of the next step you can do. This difference means that for one string, in a non-deterministic automaton, several paths are possible, ie, you have a tree of paths. Consequently, the validity of the string has to be checked against every possible path from the root node to a leaf in the tree. If at least one path leads to an accepting state, then the string is accepted.

For example, you could read the language  $L$  generated by  $G_1 = G_2$  with the following non-deterministic finite automaton (NFA):



or, equivalently, with the following deterministic finite automaton (DFA):



NB: Try to understand why the first image is an NFA, while the second is a DFA. The DFA was derived from the NFA, using the algorithm described [here](#); there exist alternative algorithms, which you should search for on your own. :)

NB: The theorem which states that a language is regular if, and only if, it is recognized by a finite automaton, is called "*Kleene's Theorem*" (in the Theory of Computation), aka "Kleene's finite automaton characterization theorem", aka "Kleene's Theorem for regular languages" (and should not be confused with "Kleene's recursion theorem", nor with "Kleene's fixed point theorem").

# Chapter X

## Exercise 05 - Example regular language acceptor

<b>Δ</b>	Exercise : 05
Example regular language acceptor	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(n)$	
Maximum space complexity : $O(n)$	

### X.0.1 Goal

The goal is to write a function which can recognize whether a given word is valid for a given regular language.

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

### X.0.2 Instructions

Let  $\Sigma = \{a, b\}$  be an alphabet. Let  $L$  be the language over  $\Sigma$  defined by the regular expression  $ab(a|b)aaa(a|b)^*ab$ .

The function must recognize whether a given string is valid for the language  $L$ . The function can only use at most 1 variable: an int to iterate over your input string. Hint: you will probably need to figure out the DFA for this regular expression on paper before implementing it.

The prototype of the function to write is the following:

```
fn my_rg_acceptor(s: &str) -> bool;
```

# Chapter XI

## Exercise 06 - Finite automaton

<b>Δ</b>	Exercise : 06
	Finite automaton
	Allowed mathematical functions : <b>None</b>
	Maximum time complexity : $O(n)$
	Maximum space complexity : $O(n)$

### XI.0.1 Goal

The goal is to write a function which builds a non-deterministic finite automaton from a given regular expression.

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

### XI.0.2 Instructions

Let  $A$  be an alphabet containing alphanumeric and whitespace ASCII characters, and  $R$  an alphabet of our regex symbols:  $R = A \cup \{*, +\}$

The function must take a regex expression and return a directed graph corresponding to a finite automaton which represents the regular expression. It is expected to provide some way to display your directed graph for the corrector.

Only the special behavior of the Kleene operators ( $*$  and  $+$  symbols) have to be supported.

The prototype of the function to write is the following:


```
fn build_nfa(regex: &str) -> FiniteAutomaton;
```



FiniteAutomaton is a structure which contains the directed graph.

# Chapter XII

## Exercise 07 - Deterministic finite automaton

	Exercise : 07
Deterministic finite automaton	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(2^n)$	
Maximum space complexity : $O(2^n)$	

### XII.0.1 Goal

The goal is to write a function which takes a non-deterministic finite automaton and converts it into a deterministic finite automaton.

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

### XII.0.2 Instructions


The prototype of the function to write is the following:

```
fn nfa_to_dfa(nfa: FiniteAutomaton) -> FiniteAutomaton;
```



# Chapter XIII

## Exercise 08 - Deterministic finite automaton minimization

	Exercise : 08
Deterministic finite automaton minimization	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(n * \log(n))$	
Maximum space complexity : $O(n * \log(n))$	

### XIII.0.1 Goal

The goal is to write a function which minimizes the given deterministic finite automaton.

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

### XIII.0.2 Instructions

The prototype of the function to write is the following:

```
fn minimize_dfa(dfa: FiniteAutomaton) -> FiniteAutomaton;
```


If the given finite automaton is non-deterministic, the behaviour is undefined.



If you want to do things in a cleaner manner, you can create a pair of type aliases, `'NFA'` and `'DFA'`, over the `'FiniteAutomaton'` type; or better, if your language is strongly-typed enough, you can use a strict alias (such as the "Newtype" pattern in Rust). This can avoid the undefined behavior above.

# Chapter XIV

## Exercise 09 - Regex matching

	Exercise : 09
Regex matching	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(2^n)$	
Maximum space complexity : $O(2^n)$	

### XIV.0.1 Goal

The goal is to write a function which tells whether a string is valid for an arbitrary regex.

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

Only the alphanumeric and whitespace ASCII characters are considered. Only the regex operators `*` and `+` are considered.

The prototype of the function to write is the following:

```
fn is_str_valid_for_regex(regex: &str, s: &str) -> bool;
```

The function must tell whether the given regex matches the string `s`.



If you understood what you did in the previous exercises, you must know by now that you should reuse 3 functions you've already written. Note: the complexity is  $O(2^n)$  only because of the nfa-to-dfa conversion. The actual 'matching a RegEx' part, when provided with a DFA, should be in  $O(n)$ .

Note that you may also implement a much faster arbitrary regex matching algorithm, based on pure strings, but it isn't required at this point.

# Chapter XV

## Interlude - Higher-level constructs of the Chomsky hierarchy

### XV.1 Limitations of finite automata

Since the state of finite automata is only represented by a "point moving along a directed graph", it is not sufficient if we want to parse anything that would require things like long-term memory, or recursion.

This is the reason why [HTML cannot be parsed using only regular expressions](#).

In the following sections and exercises, we will explore other types of automata which can alleviate this problem.

### XV.2 Type-2: Context-free grammars and pushdown automata

Like regular grammars, a context-free grammar  $G$  is formally defined by the 4-tuple  $G = (V, \Sigma, P, S)$ , where:

- $V$  is an alphabet (finite set) of nonterminals.
- $\Sigma$  is an alphabet (finite set) of terminals.
- $P$  is the set of production rules (aka "rewrite rules"). It is a finite relation (a subset of the cartesian product of two finite sets; ie, it is a set of 2-tuples) between the sets  $V$  and  $(V \cup \Sigma)^*$ , where the asterisk represents the Kleene star operation.
- $S$  is the start variable (or start symbol), used to represent the root of possible words that can be created. It is always an element of  $V$ .

What makes context-free grammars special, compared to regular grammars, is that their production rules are less restrictive. They are of the form  $A \rightarrow \alpha$  where  $A \in V$  (ie, it's a single nonterminal symbol), and  $\alpha \in (V \cup \Sigma)^*$  (ie,  $\alpha$  corresponds to *any* finite string of terminals and non-terminals, potentially the empty string).

We then have the acceptors for context-free grammars, called pushdown automata. The key aspect that differentiates a pushdown automaton from a finite automaton is the addition of (long-term) memory, through the use of a **stack**. A stack (aka LIFO, for "Last-In, First-Out") is a fundamental data structure in algorithmics, computer engineering, and language theory, since it is, in a sense, the minimal way to model computer memory. In language theory, the stack is a simple data structure that can store symbols from the alphabet  $\Sigma$ ; like stacks in the rest of computer science, it has 3 operations: "push", which adds a symbol to the top of the stack; "pop", which removes a symbol from the top of the stack; and "view" which inspects the value of the topmost element in the stack.

In the introduction, we gave an in-depth example a context-free grammar and its pushdown automaton, that we advise you to revisit now. We also advise you to look up the **Backus-Naur Form (BNF)**. It is a syntax allowing to define context-free grammars in a way that can be useful in practice. You might also want to look up **Abstract Syntax Trees**, which provide a way to visualize grammars.

## XV.3 Type-1: Context-sensitive grammars and linear-bounded automata

Context-sensitive grammars are defined by the same 4-tuple as the other, simpler grammars. Similarly to context-free grammars, it is their production rules that makes them special. The rules can distinguish from different uses of the same non-terminal  $A$ , depending on the context of  $A$  (what other symbols surround it).

Production rules in a context-sensitive grammar are of the form  $\alpha A \beta \rightarrow \alpha \gamma \beta$ , with:

- $A \in V$ , ie,  $A$  is any non-terminal;
- $\alpha, \beta \in (V \cup \Sigma)^*$ , ie, they both correspond to *any* finite string of terminals and non-terminals (potentially empty)); and
- $\gamma \in (V \cup \Sigma)^+$ , ie,  $\gamma$  is any *non-empty* string of terminals and non-terminals.

Type-1 acceptors, called linear-bounded automata, are surely easier to understand and visualize. They transform the stack into a "linearly-bounded **tape**". You can understand this as a fixed-size array of at most  $n$  values, where  $n$  is the size of the input string to the automaton. A readhead constantly points to a given cell in the tape. This provides more leeway when managing what's stored in memory and the state of the system overall (since "push" and "pop" are pretty limited). At each state transition, one can choose to move the readhead and/or write some symbol to the tape, which provides the "context" for execution.

NB: Checking whether a language truly needs a Type-1 (rather than a Type-2) grammar in order to be parsed is pretty technical, and requires the use of [Ogden's lemma](#).

## XV.4 Type-0: Recursively enumerable grammars and Turing machines

The final type of grammar is referred to as "recursively enumerable grammars" or "unrestricted grammars". Their definition is quite technical, and probably not worth your time at this point. In fact, it is exceedingly rare (outside of academic study) to find a language that actually needs a recursively enumerable language in order to be parsed. However, most modern programming languages can implement a recursively enumerable language: they have a Type-0 level of expressivity.

Once again, Type-0 constructs are better understood via their acceptors: Turing machines, named after Alan Turing. Turing machines are almost exactly like linear-bounded automata, except that their tape is understood to be potentially infinite. Note that in practice, it is never the actually infinite, but that something like "a megabyte of RAM" is often "close enough to infinite" for most applications.

Formally, a Turing machine is defined as  $(Q, \Gamma, b, \Sigma, \delta, q_0, F)$ , where:

- $Q$  is the set of states of the machine
- $q_0 \in Q$  is the initial state of the machine.
- $F \subseteq Q$  is the set of final states (the states in which the machine stops executing)
- $\Gamma$  is a set of tape symbols
- $b \in \Gamma$  is the blank symbol (the default symbol in every cells)
- $\Sigma \subseteq \Gamma$  is the set of input symbols.
- $\delta : (Q \setminus F) \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$  is a **partial function** over the set of non-terminating states. This function takes as arguments the current state of the Turing machine  $Q$  (so long as it isn't a final state) and the set of tape symbols  $\Gamma$ , then returns the new state of the machine, the new set of symbols on the tap and a direction in which the machine's head is moving on the tape ( $L$  stands for "left" and  $R$  stands for "right").

The **Brainfuck** language is a very good representation of how a Turing machine operates.

A Turing machine is powerful enough to simulate *any* other "kind of Turing machine" (anything that can implement recursive enumerable languages); this property is called **Turing completeness**, or **computational universality**.



A programming language (or something else) is said to be **Turing-complete** if it is able to simulate a Turing machine. If it can simulate (if you can implement) a Turing machine in your language, then your language is "at least as expressive" as a Turing machine.

Turing completeness is an interesting property since it allows to simulate any program inside of another program. This is especially useful when it comes to creating virtual machines for example. If you've ever seen someone [recode Flappy Bird from inside Super Mario World](#), then you have Turing completeness to thank for that !

# Chapter XVI

## Exercise 10 - Example context-free language acceptor

<b>Δ</b>	Exercise : 10
Example context-free language acceptor	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(n^2)$	
Maximum space complexity : $O(n^2)$	

### XVI.0.1 Goal

The goal is to write a function which can recognize whether a given word is valid for a given context-free language.

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

### XVI.0.2 Instructions

Let  $\Sigma = \{a, b, c, d\}$  be an alphabet. Let  $L$  be the language of palindromes over  $\Sigma$ .

The function must recognize whether a given string is valid for the language  $L$ . The function can only use at most 2 variable: an int to iterate over your input string, and a stack which can only contain values of type  $\Sigma$  (ie, symbols of the alphabet) for memory.

**You *cannot* use the length  $n$  of the input string as a guide (see hint).**

The prototype of the function to write is the following:

```
fn palindrome_acceptor(s: &str) -> bool;
```



The context-free languages that can be described by a deterministic pushdown automaton (DPDA) are a proper subset of the set of context-free languages, called "deterministic context-free languages". This means that, unlike regular languages, not all context-free languages can be parsed by a deterministic automaton. The language in this exercise, in particular, is not deterministic. This means that before returning "false", your acceptor will need to work *non-deterministically*. It will need to test all possible "routes" until it reaches a contradiction on each, and then can decide that it has tested all routes... Note that some specific contradiction sometimes imply that you'll "always get a contradiction" for all following routes.

# Chapter XVII

## Exercise 11 - Example context-sensitive language acceptor

<b>Δ</b>	Exercise : 11
Example context-sensitive language acceptor	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : $O(\exp(n))$	
Maximum space complexity : $O(\exp(n))$	

### XVII.0.1 Goal

The goal is to write a function which can recognize whether a given word is valid for a given context-sensitive language.

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  $\Sigma = \{a, b, c\}$  be an alphabet. Let  $S = \{X, Y, Z\}$  be an alphabet of control symbols. Let  $L$  be the language over  $\Sigma$  defined by the regular expression  $a^n b^n c^n$ .

The function must recognize whether a given string is valid for the language  $L$ . The function can only use at most 2 variables: an int to iterate over your input string or tape, and the tape itself (a fixed-length array of size  $n$ , where  $n$  is the length of the input string) which can contain values of type  $\Sigma \cup S$  (ie, any symbol from one of the two alphabets) for memory.

The prototype of the function to write is the following:

```
fn my_csg_acceptor(s: &str) -> bool;
```

# Chapter XVIII

## Interlude - Kleene's operators and category theory

### XVIII.1 Reminders on category theory

Those of you who remember the interlude on category theory and algebraic structures (towards the end of the module on set theory) will be interested to learn that the Kleene star and the Kleene plus are *very* important mathematical objects, in the context of category theory.

But first, some reminders are in order.

We call an **algebraic structure** a set  $S$  which is provided with an internal binary operation (a function which takes two elements of  $S$  as input, and returns an element of  $S$  as output; and is often represented as  $\star$  abstractly, or  $+$  or  $\times$  for concrete cases like addition, concatenation or multiplication), and for which this binary operation must respect some properties. Typically, these properties are some of the (non-exhaustive) following: closure, associativity, identity, invertibility/symmetry, and/or commutativity.

In the previous module, we had seen the example of a group  $G$ , where the properties of closure, associativity, identity and invertibility had to be verified by the operator.

We call a **magma** the pairing of a set and a *closed* binary operation.

We call a **semigroup** a magma with *associativity*.

We call a **monoid** a magma with *associativity* and *identity*.

We call a **group** a magma with *associativity*, *identity*, and *invertibility*.

These are examples of "kinds" of algebraic structures.

The collection of "all algebraic structures of the same kind" is called a **category**. *Set* is the category of all sets, *Sgp* the category of all semigroups, *Mon* the category of all

monoids, and  $\mathcal{Grp}$  the category of all groups. There are many, *many* more categories in mathematics; these are just a sample.

As programmers, you can think of a category "a generic type which extends a given type"; they are a collection of types with shared behavior. Phrased another way, if a given algebraic structure (aka, a given "mathematical space") is a type, then a category is a "type (a mathematical space) of types (of other mathematical spaces)". In Rust, a category could be understood as the collection of types that implement a given "trait".

## XVIII.2 Kleene's special operators seen as functors

You can map categories to other categories via something called a **functor**. These are objects which map collections of mathematical spaces to other collections of mathematical spaces.

The Kleene plus and the Kleene star are examples of functors. The Kleene plus is called the "free semigroup functor", it's a functor of  $\mathcal{Set} \rightarrow \mathcal{Sgp}$ . The Kleene star is the "free monoid functor", it's a functor from  $\mathcal{Set} \rightarrow \mathcal{Mon}$ .

The reason why the Kleene plus and the Kleene star are important is the adjective "free". A free functor can be understood as "the universal, standard way, of transforming a given algebraic structure into a minimal algebraic structure with more properties". It's a powerful, generic, mathematical "algorithm" for transforming structures into other structures. You can see how  $B = \{a, b\}$  is a set, without anything special. Feed that set  $B$  to the "free monoid functor", the Kleene star, and you get the language  $B^* = L_B = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ , which, along with the concatenation operator, forms a monoid (try to add three strings to show associativity; it's also clear that  $\epsilon$  is your identity element, and generally, you have neither inverses nor commutativity). The output of the Kleene star will necessarily be (at least\*) a monoid, no matter the set given as input. This is very powerful, abstract behavior.

The opposite (technically, "[adjoint](#)") of a "free" functor is called a "forgetful" functor. It boils down to "keeping all of the elements of the structure, but removing some of its properties". For example, we tend to write the forgetful functor from any category  $\mathcal{C}$  back to  $\mathcal{Set}$  as  $|\cdot|$ . So  $|L_B|$  is the set  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$ , but where there is no concept of concatenation whatsoever.

## XVIII.3 Going further: the free group functor

Another example is the "free group functor", which we'll call  $\mathcal{G}$ . It's very similar to the Kleene star, but it also creates an "inverse letter" for every letter. Eg, remarking that two side-by-side letters that are inverses of each other cancel out, we have:

$$\mathcal{G}(B) = \{ \begin{array}{l} \epsilon, \\ a, \quad b, \quad a^{-1}, \quad b^{-1}, \\ aa, \quad a^{-1}a^{-1}, \quad ab, \quad a^{-1}b, \quad ab^{-1}, \quad a^{-1}b^{-1}, \\ ba, \quad b^{-1}a, \quad ba^{-1}, \quad b^{-1}a^{-1}, \quad bb, \quad b^{-1}b^{-1}, \\ aaa, \quad \dots \end{array} \}$$

Fun fact: the reason I wrote "(at least) a monoid" above is because there is a special case, for the set with a single element. If we take the Kleene star  $A^*$  of  $A = \{a\}$ , we get  $A^* = \{\epsilon, a, aa, aaa, aaaa, \dots\}$ . Why is this special? Well, you'll notice that unlike in other cases, here, you also have commutativity:  $aa + aaa = aaa + aa = aaaaa$ . In fact, this space  $A^*$  is isomorphic (equivalent, as an algebraic structure) to  $\mathbb{N}$ , the natural numbers! We write this fact  $A^* \cong_{\mathcal{CMon}} \mathbb{N}$  (where  $\mathcal{CMon}$  is the category of commutative monoids). The set  $A^*$  corresponds, so to speak, to the natural numbers represented in the "unary" basis: the basis in which you count elements one-by-one, like the dots you can see on game dice or on playing cards from ace (1) to 10.


$$\begin{array}{ccccccc} aa & + & aaa & = & aaa & + & aa & = & aaaaa \\ 2 & + & 3 & = & 3 & + & 2 & = & 5 \end{array}$$

Similarly, the free group functor applied to the single element set  $A$  gives a space which is "more than just a group". It is isomorphic to  $\mathbb{Z}$ , the (additive, commutative) group of (signed) integers. Here, for example,  $a^{-1}a^{-1}a^{-1} \cong -3$ .



# Chapter XIX

## Exercise 12 - Bonus: An intermediate-level RegEx engine

	Exercise : 12
An intermediate-level RegEx engine	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : N/A	
Maximum space complexity : N/A	

### XIX.0.1 Goal

The goal for this exercise is to write an intermediate-level RegEx engine.

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

### XIX.0.2 Instructions

Let  $\Sigma$  be the printable ASCII alphabet. You must write a RegEx engine over this alphabet with the usual modern features, but only those that are of Type-3 (ie, no capture groups, variables, etc). This means that you must write a function which finds a given regex in an input string; it should return, for every index at which the string appears, a pair of the beginning and end index of the string.

The required features include:

- finding all valid instances of the string (if two instances overlap, the algorithm

should only return the first)

- implementing the Kleene star and Kleene plus, as well as the "?" operator (note that any variable length match should match maximally)
- escaping operators with the backslash "\" operator
- implementing priority via the parenthesis operator
- implementing the various ways to express union/character sets via the pipe "|", square brackets "[]", and dash "-" operators.
- implementing the "[^...]" operator to negate a character set
- implementing the "\\b" operator for start-of-word and end-of-word recognition
- implementing the "^" and "\$" operator for start-of-line and end-of-line recognition
- implementing the "\\s" operator for the whitespace charset
- anything else that you think would be nice to add after looking up what these commonly-used symbols mean: you've come this far after all, why not go even further beyond ? :)

The prototype of the function to write is the following:

```
fn regex_find(input_str: &str, regex_query: &str) -> Vec<(usize, usize)>;
```

# Chapter XX

## Interlude - another important model of computation

### XX.1 The $\lambda$ -calculus

We mentioned in our introduction that there were alternative ways to describe computation. One of them was created by Alan Turing's PhD advisor, the mathematician Alonzo Church in 1930. Turing and Church worked closely together for many years. Church's theory of computation, described through a model called the  $\lambda$ -calculus (lambda calculus), is the foundation of **functional programming**, and of a lot of constructs in general programming.

The  $\lambda$ -calculus is constituted of only 3 kinds of terms:

- Variables: Denoted as a simple letter (example:  $x$ )
- Abstractions: A function definition, taking only one argument. It is denoted  $\lambda x.M$ , where  $M$  is the name of the function. You can generally understand this as  $(x \mapsto M)$ , the function which takes  $x$  as input and returns  $M$ .
- Application: A function call. Denoted:  $(MN)$ , where  $M$  and  $N$  are lambda terms (a variable, abstraction or application)

#### XX.1.1 Currying

In  $\lambda$ -calculus, everything is built from functions. Functions can return functions, and take functions as arguments. Let's play with an important concept of  $\lambda$ -calculus, to give you an idea of the theory's *feel*.

Oftentimes, it is necessary to take a function with multiple arguments, and turn it into a function with less arguments (or a single argument). Other times, it can be helpful to go the other way around. To do so, one can use an operation called "**currying**" (or

"**uncurrying**" for the other way around). It is named in reference to Haskell Curry (yes, the Haskell programming language is named after him too).

"Currying" takes a function with  $n$  arguments as input and turns it into a function that takes 1 argument as input, but returns another function, one with  $n - 1$  arguments as input. Currying is generally repeated  $n$  times to create a chain of functions that each take a single argument, and return a function which takes as input a single argument. Then, each function can each be called in sequence with a single argument, and the overall result will still give an equivalent computation. The syntax of a few functional programming language (like OCaml) make heavy use of curried functions in how functions are naturally defined; this is often most visible when first learning to read the error output of the compiler of these languages.

For example, let  $f = (x, y) \mapsto x^2 + y^2$  be a function (say,  $f \in (\mathbb{R}^2 \rightarrow \mathbb{R})$ ). In  $\lambda$ -calculus, it can be written  $\lambda xy. x^2 + y^2$

Translated in Rust lambdas, it gives:

```
| x, y | {
  x * x + y * y
}(x, y)
```

Now, if we apply currying, it gives the function  $g = (x \mapsto (y \mapsto x^2 + y^2))$ . This function is a member of the set  $\mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$ . In the  $\lambda$ -calculus, this function can be expressed as  $\lambda x. \lambda y. x^2 + y^2$ ,

Translated in Rust lambdas, it gives:

```
| x | {
  | y | {
    x * x + y * y
  }
}(x)(y)
```

The  $x$  variable stays useable in the inner lambda expression because it is captured from the parent scope. Feeding a specific value  $x$  variable to the curried function is called a **partial application** of the function  $f$ . For example,  $g(5) = (y \mapsto 25 + y^2)$ .

Note that currying also plays an important role in category theory. Roughly put, in a frequent kind of category called a *closed monoidal category*, there is a natural isomorphism (a strong kind of equivalence) between functions which are elements of  $(A \otimes B) \rightarrow C$  and those that are elements of  $A \rightarrow (B \rightarrow C)$ . Here,  $A$ ,  $B$  and  $C$  are three structures in our category; and  $\otimes$  is the categorical tensor product, which is a generalization of the cartesian product of sets (ie, it makes two structures into the structure of 2-tuple with as the first coordinate being an element the first structure, and the second coordinate being an element of the second structure).

## XX.2 The Church-Turing thesis

One of the most famous conjectures in computer science tries to describe what "computation" is; it is called the Church-Turing thesis. This thesis comes from a time where something much less obvious than it is today was being debated: whether machines could potentially think as humans.

One specific task we wanted machines to do is run mathematical computations, to figure out the results of mathematical functions. To force a machine to do things like a human, we painstakingly figured out the steps that a human should take to compute this function; steps small and precise enough that we could have a machine do them: this was the birth of the modern branch of algorithmics. We realized that we need a couple of things for the machine to be able to run any algorithm like a human, in particular memory and iteration (sometimes bounded, sometimes unbounded). It is why such mathematical models for an algorithmic machine, the aforementioned Turing machine and  $\lambda$ -calculus, were invented.

The Church-Turing thesis basically boils down to the idea that "the (arithmetic) functions effectively computable by the human mind, *via* the use of formal mathematics, are those that are computable by a Turing machine, or equivalently, those that can be expressed and computed in the  $\lambda$ -calculus".

To this day, the Church-Turing thesis has never been formally proven. And yet, all algorithms we have ever invented or discovered could be run on a Turing machine (or a Turing-complete machine in general). Additionally, all "hyper-Turing" models (Type-"minus 1" models, so to speak) that we've tried to invent were either inconsistent, or could be simulated by a regular Turing machine. In this sense, Type-0 structures seem like the "computational limit" of our universe, and potentially all universes.

A strange consequence of the Church-Turing thesis being true is that it would go both ways: if the Church-Turing thesis is true, and there exists an algorithm that cannot be executed by a Turing machine, we would not be able to find or describe it...

# Chapter XXI

## Exercise 13 - Bonus: A small $\lambda$ -calculus compiler

<b><math>\Lambda</math></b>	Exercise : 13
A small $\lambda$ -calculus compiler	
Allowed mathematical functions : <b>None</b>	
Maximum time complexity : N/A	
Maximum space complexity : N/A	

### XXI.0.1 Goal

The goal is to write a small  $\lambda$ -calculus compiler, able to compile a program into **Brainfuck**.

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

### XXI.0.2 Instructions

You have to write the program **ft\_lambda** which takes as parameters a file containing the program:

```
./ft_lambda myprogram.lambda
```

The first step is to parse the syntax of the language.

Let:

- $A = \{a, b, c, \dots, x, y, z, A, B, C, \dots, X, Y, Z, \_ \}$  be the main alphabet.
- $s$  be an alphabet containing only whitespace characters (space, tab and newline).
- $l = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  be the alphabet of literals.
- $o = \{+, -, \times, /, \%\}$  be the alphabet of operator symbols.

Let  $G = (N, \Sigma, P, S)$  be the grammar of the language.  $P$  is defined as:

$$\begin{aligned}
 v &\rightarrow A^+ \\
 \alpha &\rightarrow vs^*v \\
 e &\rightarrow (((\{s^*\} \cup \epsilon)(l^+ \cup v \cup \alpha \cup (es^*os^*e))((s^*\{ \} \cup \epsilon)s^* \\
 \lambda &\rightarrow vs^*\{=\}s^*\{|\}s^*vs^*\{.\}s^*e \\
 S &\rightarrow \lambda^+\alpha
 \end{aligned} \tag{XXI.1}$$

Where,

- $v$  is a variable
- $\alpha$  is an application
- $e$  is an expression
- $\lambda$  is a function



This syntax might be pretty hard to grasp. You may want to try to translate it into Backus-Naur Form (BNF) first.

A program is a set of lambda functions followed by one application to start the program.

The following builtin functions must be present:

- **in**: Ignores the argument and evaluates as a byte read from the standard input
- **out**: Writes the argument byte to the standard output

The instructions set of the Brainfuck language is the following:

character	description
>	moves the tape right
<	moves the tape left
+	increments the value at the head by one
-	decrements the value at the head by one
.	writes the byte at the head to the standard output
,	reads a byte from the standard input and writes it at the head
[	if the byte at the data pointer is zero, jump forward to the matching ], else jump to the next instruction
]	if the byte at the data pointer is nonzero, jump backward to the matching [, else jump to the next instruction

Every other character is ignored.

You may find example programs and a brainfuck interpreter in the attachments.



# Chapter XXII

## Interlude - A final digression: universal cellular automata

### XXII.1 Conway's Game of Life

The Game of Life is a cellular automaton created by Horton Conway in 1970. It is a zero-player game which works with the following rules:

- The game is an infinite 2-dimensional orthogonal grid of square cells.
- Each cell can either be alive or dead.
- At each timestep, the game performs the following transitions:
  - Any live cell with fewer than two live neighbours dies.
  - Any live cell with two or three live neighbours stays alive.
  - Any live cell with more than three live neighbours dies.
  - Any dead cell with exactly three live neighbours becomes alive.

These rules somewhat model the idea that "having too little or too much population" is bad for an ecosystem. The initial state of the game can be defined by the user.

For the curious, you can very well define cellular automata with any number of dimensions, any sort of neighborhood for cells, any sort of update rules for neighborhoods, and any amount of colors for values that cells can take. In fact, most voxel-based games use (3D) cellular automata to accomplish the procedural generation of their environments.

It turns out that the Game of Life, a relatively simple cellular automaton, is Turing-complete, meaning it can be used to simulate a Turing machine, which can then execute any Turing-complete program, including the Game of Life itself (thereby creating some sort of "universe-level recursion of the universe" if you wish). "Now what does *that* look like ?", you're probably asking yourself.

So as a reward for completing this module into the guts of the mathematical theory of programming, here's the video on universality in cellular automata that was teased in the intro: [enjoy!](#)

## Contact

You can contact our association, 42AI, by email: [contact@42ai.fr](mailto:contact@42ai.fr)

You are also welcome to join the association on [42AI slack](#) and/or apply to [one of the association's teams](#).

## XXII.2 Acknowledgements

This subject is the result of collective work, we would like to thank:

- Tristan Duquesne - [tduquesn@student.42.fr](mailto:tduquesn@student.42.fr)
- Luc Lenôtre - [llenotre@student.42.fr](mailto:llenotre@student.42.fr)

This work is licensed under a [Creative Commons](#) “[Attribution-NonCommercial-ShareAlike 4.0 International](#)” license.

