Portfolio
Models of Computations

Alberto Defendi
alberto.defendi@helsinki.fi

May 21, 2022

Contents

1 Introduction

Why studying models of computations?

Studying models can help us to develop a clear understanding of the basis of computer science. Since its foundation, computer science has evolved quickly; staying updated on the cutting-edge technologies was, and is, part of the profession of computer scientists. For the working CS person, knowing the fundamental ideas of their field reduces the gap between the tools of today, and the technologies of tomorrow. In fact we will see many abstract theories that are dynamic and the beating hearth of CS, such as Section 2. Some applications are listed in Section 2.4. Models gives a view of theoretical computer science. The subject are many connections with the foundations of mathematics, and the theory of algorithms. Some examples are the pigeonhole principle in Section 2.3.1, or the powerset in Section 2.2.

2 Regular languages

A language $\mathcal{L}$ is a set that can be generated by a rule. We can represent every language as a set, using the set-builder notation. For example $\mathcal{L} = \{a^n | n \geq 0\}$, where a^n means the repetition of a-symbols, generates $\{aaaaa\}$ for $n = 5$. The language we defined above, let us generate an infinite amount of strings, such as $aaaa, aaa, a, a^0 = \varepsilon$. This process of generating the possible strings from the set happens in the mind[1], but can we teach a computer to generate and recognize strings with that rule? The beautiful theory of regular languages tries to answer this question.

Theorem 1 (Regular language)**.** A language is regular, if it can be recognized by a finite state machine.

Generally, a language is regular if satisfies one of the Chomsky Normal Forms, that means that the language is composed by terminal symbols, has a start variable, and allows composition of variables.

2.1 Drawing regular languages

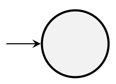
We know that a language is regular if it can be represented by a *finite-state machines*, also called automata. Intuitively, an automata models states and transitions in a system; the book (*1*) provides visual examples about what can be modeled with automata. In this part we will refer to automata as *deterministic finite automata* (DFA), later we will introduce *non-deterministic finite automata* (NFA). The difference will become clear in 2.2. Let us draw all the points in the definition to get a visual intuition. Finite state machines are This step is important, as it will show how automata and regular languages are connected, and is asserted by the following theorem.

Theorem 2. (Regular language) A language is regular if and only if can be accepted by a finite state machine.

Note. To prove a statement of the form $A \Rightarrow B$, we assume A, and work towards reaching B using forward reasoning. We apply the same idea when we prove in two directions. The operator $\Leftrightarrow \equiv \Rightarrow \wedge \Leftarrow$, so divide the proof in $A \Rightarrow B$ and $B \Leftarrow A$.

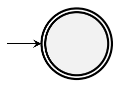
Let us now see the idea for the $\Rightarrow$ direction. We can prove it using the definition of regular language.

Proof. Is $\emptyset$ a regular language? Yes, it is an automata with a start without an accepting state.



[1]But remember that the set is alive in the world of mathematics.

Is ε a regular language? Yes, it is an automata with only an accept state.



The most complex cases are 4, 5, 6. For Points 4 and 5, you can take two automata N_1, N_2 and form a new automata from the union (or composition of the two). □

Note. Observe that points 1, 2, 3 follow directly, and points 4, 5, 6 follow because we proved that a regular language is closed under union, composition, and star operation. We can show that regular expressions are closed under many other properties, such as shuffle, rotation, reverse, and homomorphism. In contrast, as we will see, context-free languages have less properties.

Some facts about automata:

- For any language, there is an infinite number of automata that recognizes it.
- It becomes easier to draw a DFA if you see it as a directed graph $G = (V, E)$, with each state q_i in the nodes, and each image of the transition function $\delta(q_i)$ as the edges $e(u, v)$ of the nodes.
- Automata can be minimized. You can do it by simplifying the regular expression using the algebraic rules of the operations, or also graphically.

2.2 Difference DFA and NFA

Let us try to compare the two definitions of DFA and NFA. If we look closer, they share many similarities. Observe how we define Ran(δ).

DFA $\delta : Q \times \sum \to Q$

NFA $\delta : Q \times \sum \to \mathcal{P}(Q)$

This can explain why DFA only have exactly one transition arrow for every state, and one exiting state. In contrast, NFA have one or more transition arrow for each state, and can terminate in more exit states.

We can conclude that NFAs are a generalization of DFAs, because every powerset has a set generating it (by the powerset axiom), every DFA has a corresponding NFA (Theorem 1.39). We are going to see this pattern again with other languages of the universe.

2.3 Spotting non-regular languages

Are there forms of languages that are non-regular? See that most of the languages of the form $\{0^n 1^n \mid n \geq 0\}$ are non-regular, and that those of the form $\{(01)^*\}$ are regular, but using the pumping lemma provides formal evidence. It is interesting to see[2] that regular languages cannot be infinite, thus there must be a larger class accepting those languages (we will see this later in CFLs).

[2]https://cs.stackexchange.com/questions/47835/can-a-regular-expression-be-infinite

2.3.1 Pumping lemma

The pumping lemma, is a powerful device to prove that a regular language is non-regular. This theorem can be proved using the *Pigeonhole principle*[3]. Intuitively, if we have p pigeons, and less than p boxes there will be a box with at least two pigeons. We will not re-state the theorem here, but we provide an algorithm that can be used to prove that an expression R is not regular, trough a contradiction.

Note. This method is called *reductio ad absurdum*, because we assume B and try to reach (reduce) A.

Proof. The common steps are,

1. Assume that A is regular.
2. Let p be the pumping length, let w be a partition of A, such that $w = xyz$, and $|w| \leq p$.
3. Now check each case of the pumping lemma.
   (a) For each $i \geq 0, xy^i z \in A$.
   (b) $|y| > 0$
   (c) $|xy| \leq p$

   Step. 3 is crucial for the contradiction, here we will show that increasing (or decreasing) the i, the resulting string is not in A. This is a contradiction, and the language is non-regular.
4. Next, we provide the evidence: For every legal decomposition of w into xyz, obeying $|y| \geq 1$, $|xy| \leq p$, there exists $i \geq 0$ such that $s' = xy^i z \in A$. But this is a contradiction because $s' \notin A$. Thus A is not regular.

$\square$

Note. Choosing the partition (step 2) sometimes requires more creativity, and can cause confusion to the reader. To make it easy:

1. Generate a long enough string $s \in A$.
2. Divide s in xyz. Write it down and chose suitable segment. Make a wise decision for y because it will be the character that will be pumped up or down later.
3. Pump y up or down. It means that the string will have more or less characters in some place. Then observe if the new string is in A, or points a-c are all true.
4. If possible, draw a picture of the partition divided by each variable.

[3]we will use it at least two times in the course

2.4 Applications

Automata are widely used in computer science, examples are protocols (e.g TCP), AI, compilers, and NLP. The theory of regular language is very powerful; we could implement a simple REGEX parser based on the theory of what we just stated. Automata can also be used to model many patterns in everyday life, such as language, or games, like rock, paper, scissors. Although, regular languages have many limits, and do not suit all applications (e.g programming languages), as we will see soon with CFGs.

3 Context-free languages

We previously talked about regular languages, where showed using the pumping lemma when languages are regular or non-regular. We observed that most of the languages of the form $\{(ab)^*\}$ are regular, while languages of the form $\{a^n b^n\}$ are non-regular. But is there something that can accept languages of this form? It turns out that we can extend the class of regular languages to a more powerful device that can accept non-regular languages. These languages are called *Context-free languages* (CFL).

You can see that the definition is similar than the definition of automata, let us try to compare them.

A finite automation is a 5-tuple

$$A = (Q, \Sigma, \delta, q_0, F).$$

A context free language (CFG) is a 4-tuple

$$G = (V, \Sigma, \mathcal{R}, S).$$

Dissecting point-by-point,

1. States Q are similar to variables V.
2. Both have an alphabet.
3. Both have a transition function defined on the states.
4. The start has two different names $S = q_0$

The definition is similar to what we saw for finite automata, however, it is important to stress that automata cannot represent all CFLs (why?). To do so, we shall define push-down automata.

The construction of CFLs is composed by two process: derivation and parse trees. Let us derive an example from the definition. Let the alphabet be $\sum^* = \{a, b, \varepsilon\}, V = \{A\}$ be the variables of the language, A be the starting point. We want to compose the language of all alternating strings. We set the rule $\mathcal{R}$ to be

$$A \to Aa \mid aAaA \mid \varepsilon.$$

A derivation is the process of generating a string from a language. We start form the simple case (Aa) and work towards creating the string recursively, repeatedly applying the pattern.

$$A \Rightarrow aA \Rightarrow aaAaA \Rightarrow aaaAaaA \Rightarrow aaa\varepsilon Aaa\varepsilon.$$

After we are satisfied with the result, we can stop inserting generating, and insert a terminal symbol. We can observe that CFL always have a terminal character like $\varepsilon, \emptyset$, otherwise the string would be infinite.

Thinking about what we have just done, designing a CFG becomes easier if we think recursively, and try to build the language looking at the original expression. First look at the base case. Then try to compose the strings modeling the rule you want to infer. Always check that the string you derive is in the language.

3.1 Ambiguity

An important notion for CFL is the idea of ambiguity.

Definition 1. (Ambiguity) A context-free language is called ambiguous if it has more than one left-most derivation.

The CFL we defined above is ambiguous, because

$$A \Rightarrow Aaa \Rightarrow aAaAaa \Rightarrow a\varepsilon a\varepsilon aa \Rightarrow aaaa$$
$$A \Rightarrow aAaA \Rightarrow aAaaAa \Rightarrow a\varepsilon aa\varepsilon a \Rightarrow aaaa.$$

Knowing ambiguity we can learn the concept of left and right derivation, which is important to see what regular languages can be generated with a CFG. We can observe that a CFG generates different derivations, and different parse trees.

3.2 Relation with other languages

Regular languages (e.g $L(a^*b^*)$) are a subset of Context Free Languages (e.g $\{a^nb^n\}$) thanks to the theorem we proved in the exercises. One important theorem that we proved in the homework 3 states the connection between CFG and regular languages.

Theorem 3. A regular language R is generated by a CFG if, and only if R is regular.

A further question is: are there languages that are outside context free? The answer is yes, and it can be proven modifying the algorithm in 2.3.1.

4 What is a Turing Machine?

A Turing machine is a theoretical model for a computer, limited only in size than in power and implications on contemporary science. The first is about the feasibility of a computation. A *Turing Machine* (TM) defines what can, or cannot be computed, as we will see in *decidable* and *undecidable* problems in § 4.1. The second is about measuring speed, a TM offers a precise framework to calculate the time taken by an algorithm § 5. This motivates deep philosophical ideas; given a process as the input of a Turing Machine, it can determine whether a question can be answered or not. As the recents developements of mathematical logic show, there are some statements whose truth can never be determine, such as telling if a statement contains has an infinite loop. Knowing what questions are feasibly answerable provokingly touches the philosophy of mind, as Hofstadter in Gödel, Escher, Bach (*2*) reminds, "the brain is some sort of "mathematical" object", but can a Turing Machine be that? For the sake of analogy, we can see the brain as a finitely long cells of storage which has a function that writes inputs from the environment on the cell array, following some directions -to simplify, assume that these are some mathematical symbols of an alphabet. A brain can start an action, e.g "start reading", move -left and right- trough lines of text, and stop when the process if finished.

Or better, a Turing Machine is a finite automation (*3*) with a (potentially) infinite tape as its memory. As for automata, we are given an alphabet Σ with the difference that we operate on a second alphabet, the tape alphabet Γ that is a super-set of Σ. The powerful feature of a TM is the transition function δ, that takes a pair $\langle q_i, \gamma_i \rangle$ where $q_i \in Q$, $\gamma_i \in \Gamma$, and transforms the output as a triple $\langle q_i, \gamma_i, p \rangle$, where p is any direction given by a set of rules, for example if $p \in \{L, R\}$, the function can only go L (left) and R (right). Changing slightly this function makes possible to change the behavior of the Turing Machine, such as if we remove L, the TM will go only right!

4.1 What do Turing Machines do?

Turing machines take as input a language, and are capable to perform many mathematical operations that we can do on languages, for example sum, recursion, and difference. A TM that does these operations successfully always terminates a computation, we will say that such machine *decides* the language. Other processes cannot be performed by TM, such as determining if a language computes a recursive function that ends -these are called undecidable problems. These type of problems motivate the importance of the Church Turing thesis in the context of compatibility theory. Indeed, using Turing Machines, we can prove the two Gödel's incompleteness theorems, as Standford's Enciclopedia of Philosophy describes in detail (*4*).

The main classes of languages that are used with Turing Machines are defined as follows.

Definition 2. (Recognizable, or recursive) A language $\mathcal{L}$ if a TM can determine if a given word belongs to the language or not.

An example that scratches the idea of incompleteness is determining if a program with an infinite loop will stop its computation, this is known as the Halting problem.

The notion of Turing Decidable languages tightens this constraint to avoid infinite loops.

Definition 3. (Decidable, or recursively enumerable) A language is decidable if and only if is Turing recognizable and the TM *accepts* the string in the language and *rejects* the string not in the language.

For example, constructing the language that sums two natural numbers $\{a, b \in \mathbb{N} \mid a + b = c\}$ is acceptable, because its computation always end in an *accept* or *reject* state.

4.2 Proving Decidability

To prove that a language is decidable, we shall first introduce the idea of Universal Turing Machine. A TM U is an Universal Turing Machine, if it takes as an input another Turing Machine and a string w, and terminates in an *accept*, *reject* state. This is the great result of Church-Turing's thesis, as it allows to check decidable languages, we again refer to SEP for the implications on philosophy (*5*).

An example of simple undecidable problem, is whether a programming language can predict if a program will stop on an endless loop. In this case U is the interpreter, and the Turing Machine M is our source-code, that can be encoded in a machine-readable language. Let us prove this formally; The idea is to take two TM, a Turing machine for the Java program, and one for the Java interpreter. We try to answer the question: can the interpreter detect if there are loops in the code? The resulting proof will be simple, as we only need to pass the Turing Machine to a machine that we proved (see book) to be undecidable.

Proof. Let U be a TM representing the program accepting a Java program. Let $J_{TM} = \{\langle J, w\rangle \mid J \text{ is a TM and } J \text{ accepts } w\}$ be a TM representing the Java program. But by Theorem 4.11 J_{TM} is undecidable, therefore an algorithm cannot decide if another algorithm can halt or not. □

4.3 Turing completeness

Being a theoretical computer, Turing machines are the *model of computation* that is nowadays used as basis for computers.

Definition 4. (Turing Completeness) A language is Turing complete, it has all the properties of what can be computed by a TM.

For example, it can perform the operation that we listed in § 4.1. This is important, because showed in the course that algorithms and Turing Machines are equivalent. This implies that if given an algorithm and one or more Turing Complete languages, it can be solved using any Turing-complete language. For example, consider an algorithm that given a, b as input, computes $a + b$. Its implementation can be implemented in Python and then converted to Lisp, and vice-versa, because both are Turing complete, and because a TM that runs the algorithm can be constructed.

4.4 Determinism

As with automata, there are deterministic and non-deterministic TMs. This idea will be useful as a model of computation for computing solvable, and unsolvable problems. In the same fashion as for deterministic and non-deterministic automata, the difference between the twos relies on the transition function: Non-deterministic TMs have the range of the transition function defined on the powerset, which provides more terminating states.

References

1. M. Sipser, *Introduction to the Theory of Computation* (ACM New York, NY, USA, 1996).
2. D. R. Hofstadter, *Gödel, Escher*, 1979.
3. `https://web.stanford.edu/class/archive/cs/cs103/cs103.1142/lectures/18/Small18.pdf`, Accessed: 2022–05-08.
4. P. Raatikainen, in *The Stanford Encyclopedia of Philosophy*, ed. by E. N. Zalta (Metaphysics Research Lab, Stanford University, Spring 2022, 2022).
5. L. De Mol, in *The Stanford Encyclopedia of Philosophy*, ed. by E. N. Zalta (Metaphysics Research Lab, Stanford University, Winter 2021, 2021).
6. J. Erickson, *Algorithms*.
7. `https://www.csie.ntu.edu.tw/~lyuu/complexity/2012/20121106s.pdf`, Accessed: 2022–05-15.