Lecture Notes on

Automata and Formal Languages

(CIIC 5045)

by Jaime Seguel

Department of Computer Science and Engineering
College of Engineering
University of Puerto Rico at Mayagez

January 28, 2018

Chapter 1

Introduction

This chapter is a review of basic notions in logic and set theory, and an introduction to the main objects of study in the theory of computation.

1.1 What is Theoretical Computer Science?

Modern computers are electronic devices that implement the basic *logic operations*. Despite this seemingly simplistic set up, it is quite evident that computer devices can help solving a surprisingly large array of practical problems, far beyond of what might be expected for a machine that performs only the three basic logic operations. What are the limits of the problem solving power of these devices? Answering this question requires more than computer experimentation, it requires a full-fledged *theory of computation*.

Being binary logic operations the sole operations of computing devices, any theoretical attempt to answer this question will be naturally embedded in the theory of mathematical logic. As such, the theory of computation, or theoretical computer science, assumes all axioms and theorems of the latter.

Theoretical computer science constructs abstract models of computer algorithms, or **automata**, and of the problems that intend to solve, or **formal languages**; and studies the power of each model of automata to process the information encoded in the **strings** of a formal language. As we will see, each model of automata is tied not only to the way the strings are processed, but also to a data storage and access paradigm.

But before delving into descriptions of formal languages and automata, let's review some of the basic elements of propositional logic.

1.2 Review of Logic

Logic studies the rules of deduction.

1.2.1 Propositional Logic

The primary level of logic is propositional logic.

Definition 1.2.1. A *declarative sentence* is a sentence that declares a fact. A *proposition* is a declarative sentence that can be proved to be true or false.

For instance,

Example 1.2.1. Consider the following sentences:

1. "$x^2 - 1 = 0$" is a declarative sentence, but not a proposition, because to decide whether is true or false, it is necessary to know the value of x.
2. "Mayagez is the capital of Puerto Rico" is a declarative sentence that is a proposition, because it can be proved true or false by looking at the administrative organization of Puerto Rico.
3. "What time do you have?" is not a declarative sentence.
4. "Mayagez is the most beautiful town in Puerto Rico" is a declarative sentence, but is not a proposition since there is no way to prove it true or false.

Indeed, proving the truth value of a proposition requires a well-establish *framework of principles, and derivation rules*. For instance, in the legal arena, the Civil Code is an example of such framework, as it defines the civil crimes. Also a Procedural Code must be followed to prove the guilt of a defendant. Similarly, the axioms of Euclidean Geometry is a basic framework to prove the truth or falsity of a geometrical proposition, and the definitions and operations of set theory, the basic framework for proving propositions about mathematical sets. The proof of the truth value of a mathematical proposition uses the rules of propositional logic, which are expressed in terms of connectives, propositional variables, parentheses and formulas, shown in Table 1.

connectives	$\vee, \wedge, \neg$
variables	$x_1, x_2, \ldots$
parentheses	$(,)$
formulas	$(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\neg\phi)$, etc.

Table 1: Components of Statements of Propositional Logic

In propositional logic, variables that take either true (T) or false (F) as values, this is *Boolean* variables, are used to represent propositions. It is important to keep in mind the difference between the *meaning* of a variable and its value.

Example 1.2.2. By writing

$$p =: \text{“Mayagez is the capital of Puerto Rico”}$$

we declare the meaning of p, not its value. The value is

$$p = F\,(\text{false}).$$

An assignment of a value to Boolean variable is called a *truth assignment*.

Definition 1.2.2. The logic operations, or logical connectives, are three; namely $\neg$ ("logical negation"), $\vee$ ("logical or") and $\wedge$ ("logical and"). These are defined by the next truth tables.

x_1	$\neg x_1$
T	F
F	T

,

x_1	x_2	$x_1 \vee x_2$
T	T	T
T	F	T
F	T	T
F	F	F

and

x_1	x_2	$x_1 \wedge x_2$
T	T	T
T	F	F
F	T	F
F	F	F

Table 2: Truth Tables of Statements of Logical Operations

Definition 1.2.3. A proposition is *compound*, if it contains logical connectives. Otherwise, it is an *atomic proposition*.

Example 1.2.3. Consider the next propositions,

1. p =: "Operating system and user's code operate on separated memory segments" is a compound since it includes a conjunction.

2. p =: "The moon is a satellite" is atomic.

Definition 1.2.4. A representation of a proposition in terms of variables, logical connectives, and parenthesis is called *Boolean formula*.

Example 1.2.4. Let x_1 =: The operating system runs in kernel space, x_2 =: the user code runs in user space; and x_3 =: kernel space is different from user space is expressed as

$$\phi(x_1, x_2, x_3) = x_1 \wedge x_2 \wedge x_3.$$

The value of a Boolean formula (or equivalently, a compound proposition) depends on the values of its *atomic constituents*, the order of precedence of the operators involved or the order determined by parenthesization. The process of identifying the atomic constituents and logic operators in a compound formula is called *parsing*.

Remark 1.2.1. Two other common symbols, $\oplus$ and $|$, are the "exclusive or" and "nand" binary operators expressible in terms of the first four symbols. Specifically,

$$x_1 | x_2 \equiv \neg(x_1 \wedge x_2), \text{ and}$$
$$x_1 \oplus x_2 \equiv (x_1 \vee x_2) \vee (x_1 | x_2));$$

where $\equiv$ means that the statements *are equivalent*, this is, have the same truth values on the same truth assignments.

1.2.2 Elements of Predicate Logic

In subsection 1.2.1 it was stated that "$x^2 - 1 = 0$" is not a proposition because its truth value cannot be determined without knowing the value of x. Nonetheless, is clear that "$x^2 - 1 = 0$" is a declarative statement, and it is a kind of statement that is very common in mathematics. The analysis of this type of statement falls within the next level of logical analysis, which is generically termed *predicate logic.*

Definition 1.2.5. A declarative statement with variables is called a *predicate* if its truth value can be proved.

Example 1.2.5. Let's first convert "$x^2 - 1 = 0$" into a full-fledged predicate by declaring the *domain* of the variable x. Let's say

$$P(x) =: \text{"}(x \in \mathbb{Z})\, x^2 - 1 = 0.\text{"}$$

Then, $P(1)$ is true but $P(0)$ is false.

Thus, in principle, P is a mapping defined on $\mathbb{Z}$, that takes values on $\{\mathrm{T}, \mathrm{F}\}$. This variation is eliminated with quantifiers.

Definition 1.2.6. The quantifiers are $\forall$ (universal quantifier), and $\exists$ (existential quantifier). A sentence of the form

$(\forall x \in D)P(x)$ means that P holds for all elements in D;
$(\exists x \in D)P(x)$ means that P holds for at least one element in D;
$(x \in D)P(x)$ is a free predicate.

Example 1.2.6. Some simple illustrations are:

1. $(x \in \mathbb{Z})\, x^2 - 1 = 0$ is a free predicate.
2. $(\forall x \in \mathbb{Z}) x^2 - 1 = 0$ is a false predicate.
3. $(\exists x \in \mathbb{Z}) x^2 - 1 = 0$ is a true predicate.

The values of $x \in D$ that make true a free predicate are used to define sets.

Definition 1.2.7. Let x be a variable over the domain D. Let $P(x)$ be a predicate on x. Then,

$$S = \{x \in D : P(x)\}$$

is the set of all values of x in D, for which $P(x)$ is true. $P(x)$ is called *condition of membership* in S.

Example 1.2.7.

$$S = \{x \in \mathbb{R} : x^2 - 1 = 0\} = \{-1, 1\}.$$

The rules of negation of a quantified predicate $P(x)$ are

1. $\neg(\forall x \in D)P(x) \equiv (\exists x \in D)\neg P(x)$, and
2. $\neg(\exists x \in D)P(x) \equiv (\forall x \in D)\neg P(x)$.

x_1	x_2	$x_1 \to x_2$
T	T	T
T	F	F
F	T	T
F	F	T

Table 3: Table of Values for Implication

1.2.3 Implication

Implication represents a process for the derivation of a proposition or predicate Q from another proposition or predicate P. From now on, we refer to propositions and predicates as *statements*.

Definition 1.2.8. Given two statements p and q, we say p implies q, denoted $p \to q$, if and only if *there is a* $\underline{\textit{correct process}}$ for deriving q from p.

Here again, *correct process* means a sequence of steps taken according with an accepted set of rules. For example, the procedural codes in the implication of culpability in law, or the rules of algebra, in the implication of an algebraic equation.

The truth of the existence of a correct process ($\to$) is given by the next table.

Table 3 establishes that there is a correct process to derive x_2 from x_1 in all cases, expect when x_1 is T and x_2 is F.

Example 1.2.8. Consider the statements $x_1 =:$ "$2 \geq 3$" and $x_2 =:$ "$0 \geq 0$" . A correct process to derive x_2 from x_1 is

$$\begin{aligned} 2 \geq 3 &\to 0 \cdot 2 \geq 0 \cdot 3 \\ &\to 0 \geq 0. \end{aligned}$$

This is a correct process that derives a true statement out of a false one. Now, if $x_2 =:$ "$4 \geq 5$", we may use

$$\begin{aligned} 2 \geq 3 &\to 2+2 \geq 3+2 \\ &\to 4 \geq 5; \end{aligned}$$

which illustrate the case of a correct process for deriving a false statement from a false one.

1.2.4 Biconditionals and Equvalence

Definition 1.2.9. Given two statements P and Q, we say P *if and only if* Q, denoted $P \leftrightarrow Q$, if

$$(P \to Q) \wedge (Q \to P)$$

is true.

It turns out that $P \leftrightarrow Q$ is the same as $P \equiv Q$.

1.2.5 Proof Methods

The theoretical body of a deductive discipline is mainly built though implications. In mathematics, the process of implying the truth value of one statement from another statement, is referred as a *proof*. We summarize here the main forms that a proof may acquire.

In mathematical proofs, statement x_1 is termed *hypothesis* and x_2, *thesis*. The truth value of the implication $x_1 \to x_2$ is actually used to demonstrate the truth value of the thesis. Table 3 shows that if x_1 is true, then the truth value of x_2 is the same as the truth value of the implication. This is, if there is a correct derivation of x_2 from a true x_1, then x_2 is also true. And, if there is no correct derivation of x_2 from a true x_1, then x_2 is false. Notice that if x_1 is false, table 3 does not decide the truth value of x_2. Thus, only true hypothesis are to be considered in mathematical implications.

Direct Proof

Applies to statements of the form $(\forall x \in D)P(x) \to Q(x)$. A direct proof takes $P(x)$ as a true hypothesis and use it to derive $Q(x)$.

Example 1.2.9. Statement: $(\forall n \in \mathbb{N})n$ even $\to n^2$ even

Proof. Since by hypothesis n is even, $\exists k \in \mathbb{N}$, $n = 2k$. But then,

$$\begin{aligned} n^2 &= 2k \cdot 2k \\ &= 4k^2. \end{aligned}$$

Now, $k \in \mathbb{N} \to k^2 \in \mathbb{N}$ and $2k^2 \in \mathbb{N}$. Thus, $n^2 = 2q$ where $q = 2k^2 \in \mathbb{N}$. □

Contraposition

Applies to statements of the form $(\forall x \in D)P(x) \to Q(x)$. A proof by contraposition is one that show a correct implication for the contrapositive $(\forall x \in D)\ \neg Q(x) \to \neg P(x)$ instead of the original stament.

Example 1.2.10. Statement: $(\forall n \in \mathbb{N})n^2$ even $\to n$ even

Proof. By contraposition. Let $n \in \mathbb{N}$ be odd. Then, $(\exists k \in \mathbb{N})$, $n = 2k+1$. Thus,

$$\begin{aligned} n^2 &= (2k+1)(2k+1) \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1. \end{aligned}$$

Since $k \in \mathbb{N}$, $q = 2k^2 + 2k \in \mathbb{N}$ and thus, $n^2 = 2q + 1$ is odd. □

Proof of Existence

Applies to statements of the form $(\exists x \in D)P(x)$. The proof shows the existence of a value of x that satisfies $P(x)$ either by constructing the value or giving a theoretical argument proving that such value of x must exists.

Example 1.2.11. Statement: The set $EVEN$ of all even numbers, is countable.

Proof. To demonstrate that there is a one-to-one and onto mapping between $\mathbb{N}$ and $EVEN$. Let $(\forall n \in \mathbb{N})\ f(n) = 2n$.

1. *f is one-to-one.* Let $n_1 \neq n_2 \in \mathbb{N}$. Then, $2n_1 \neq 2n_2$ and therefore, $f(n_1) \neq f(n_2)$. So, f is one-to-one.

2. *f is onto.* Let $n \in EVEN$. Then, $(\exists k \in \mathbb{N})$ such that $n = 2k$. But then, $f(k) = n$. So, f is onto $EVEN$.

□

Counterexample

Is similar to a proof of existence except that it is used to demonstrate that a statement of the form $(\forall x \in D)\ P(x)$ is false. Thus, a counterexample is a value of $x = a$ for which $P(a)$ is false.

Example 1.2.12. Statement: $(\forall x \in \mathbb{R})\ \sqrt{x}$ is irrational.

Proof. The statement is false. Counterexample: Let $x = 4$. Then, $\sqrt{4} = 2$ is a rational number. □

Contradiction

This method is also termed *reduction to absurdum*. It applies to a statement of the form $(\forall x \in D)P(x)$ and consists in assuming its negation $(\exists x \in D)\neg P(x)$ as an hypothesis and derive from it a false statement $Q(x)$. Since the derivation is correct, from the last row in the implication table 3 we conclude that the assumption $(\exists x \in D)\neg P(x)$ is false. Hence, $(\forall x \in D)P(x)$ is true.

Example 1.2.13. Consider the next statement of non-divisibility.
Statement: $(\forall n \in \mathbb{Z})(n^2+2)/4 \notin \mathbb{Z}$

Proof. By contradiction. Assume as hypothesis that $(\exists n \in \mathbb{Z})\ (n^2+2)/4 \in \mathbb{Z}$. Then, there is $k \in \mathbb{Z}$ such that $n^2+2 = 4k$. Now, if n can be either even or odd. We consider each case separately.
Assume n is even. Then, $n = 2m$ for some $m \in \mathbb{Z}$. But then,

$$\begin{aligned}4k &= n^2+2\\ &= (2m)^2+2\\ &= 4m^2+2.\end{aligned}$$

By dividing both sides by 2,

$$2k = 2m^2 + 1 = q.$$

Now, this implies that $(\exists q \in \mathbb{Z}) q \in EVEN \land q \in ODD$ which is false. Thus, the original statement is true for even numbers.
Assume n is odd. Then, $n = 2m + 1$ for some $m \in \mathbb{Z}$. But then,

$$\begin{aligned} 4k &= n^2 + 2 \\ &= (2m+1)^2 + 2 \\ &= 4m^2 + 4m + 3. \end{aligned}$$

Therefore,

$$4k - 4m^2 - 4m = 3,$$

which implies that $3 \in EVEN \cap ODD$ which is false. Thus, the original statement is also true for odd numbers. $\square$

Induction

Proof by induction apply solely to statements of the form:

$$(\exists n_0 \in \mathbb{N})(\forall n \in \mathbb{N}, n \geq n_0) P(n).$$

A proof by induction consists of two main steps:

1. Prove that $P(n_0)$ is true, for $n_0 \in \mathbb{N}$.

2. Prove the implication $(\forall n \in \mathbb{N}) P(n) \to P(n+1)$

By proving both steps, we conclude that $(\forall n \in \mathbb{N}, n \geq n_0) P(n)$ is true.

Example 1.2.14. Statement:

$$(\forall n \geq 0) P(n) =: \sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Proof. 1. *Proof that* $P(0)$ *is true.*

$$P(0) = \sum_{i=0}^{0} i = 0 = \frac{0 \cdot 1}{2}.$$

2. *Proof that* $(\forall n \in \mathbb{N}) n \geq 0, P(n) \to P(n+1)$.

$$\begin{aligned} P(n+1) &= \sum_{i=0}^{n+1} i \\ &= (\sum_{i=0}^{n} i) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)(n+2)}{2}. \end{aligned}$$

□

1.2.6 SAT and variants

Finally, let's define,

Definition 1.2.10. A Boolean formula or a predicate P is

1. a *tautology* (TAU) if it is true for all truth assignments,
2. *satisfiable* (SAT) if there exists a truth assignment that makes it true, and
3. a *contradiction* (CON) if it is false for all truth assignments.

1.3 Modeling problems

Although problems have different features and are formulated in a variety ways, they all share a *set of objects*, some *relations* between some or all these objects, and a *question* on the objects and/or its relations. A solution of a problem is an *answer* to the question.

As computers operate with binary logic, a *computed answer* to a problem can be only *True* (1); or *False* (0).

Definition 1.3.1. A problem is said to be a **decision problem** if its answer is *Yes* or *No* (*i.e.* True or False).

Remark 1.3.1. It is worth remarking that the *decision*, this is the Yes or No answer, may come as the answer to the question of whether the output of an algorithm is indeed a solution. Often, we identify with algorithm a computing procedure that returns an object, a number, a list, a set, etc and assume that the algorithm output *is the solution* of the problem. We know that the algorithm is correct (Hopefully you have proved that!) and thus, the last step of the problem solution, which is the **verification** of the computed answer, is dropped. However, this last step that you often assume or perform mentally, is to be considered part of the algorithm. For instance, if the problem is to find a root of a polynomial, a computer algorithm will most probably return a real number. This number is, technically speaking, a *candidate* root, but not yet a full fledged root. In order to verify whether the candidate solution is a solution we will most probably evaluate the number in the polynomial, and answer with a definite Yes, if the evaluation gives 0, or No, otherwise.

Example 1.3.1. Here are two decision problems:

1. Given a positive natural number n, Is n prime?
2. Is 2 a solution of $x^2 - 2x + 1 = 0$, with x real ?

In the first problem, the objects are the natural numbers, the relation is the property of being prime, and the question, *Is n prime*? clearly has only two possible answers: Yes or No. In practice, we will probably use the *Sieve of Erathosthenes* and verify if n appears in the list of prime numbers, to answer the question. In the second problem the objects are the real numbers, the relation is a quadratic equation. This is a direct verification problem and thus, the answer is also Yes or No.

It turns out that not all problems that are solvable with computer algorithms are rendered exactly as decision problems, as is the case of *optimization problems*. A solution of an optimization problem is by definition, the *best possible answer* and the question: Is this the best possible answer? cannot be answered with a Yes or No in some instances where the search space is infinite.

Question 1.3.1. Why just some instances of “infinite” search spaces? Is it **always** possible to ensure that an output is the “best answer” if the search space is finite? Justify your answer.

Optimization problems are cast as decision problems by requesting an *approximate solution* instead of the best possible solution.

Example 1.3.2. Find the roots of the equation $x^3 - 3x^2 - x - 1 = 0$, x real.

This equation has no integral roots, and thus, each computed result cannot be claimed as a *root* but as an *approximated root*. Thus, we rewrite the example as

Example 1.3.3. Find the roots of the equation $x^3 - 3x^2 - x - 1 = 0$, x real, approximated up to two significant figures.

The problem becomes a decision problem as, after applying *Bisection* we get the candidate roots $-.68, .46$, and 3.2. And after replacing in the equation we verify that the result is 0 within two significan digits.

In theory of computation, decision problems are represented by their set of solutions. This is, a problem looks like

$$S = \{v \in D : P(v)\}$$

where v is a variable, D the domain of the variable, and P a *predicate* on v satisfied if and only if v takes values that are a solution of the problem.

Example 1.3.4. The solution set of the problem in item 1 in the previous example, this is, the problem of deciding whether a natural number is prime, is:

$$PRIME = \{n \in \mathbb{N} : \forall m \in \mathbb{N}, \frac{n}{m} \in \mathbb{N} \leftrightarrow m = n \vee m = 1\}.$$

In this particular case the predicate is just the definition of prime number. As for item 2 in the same example, we have

$$SOL = \{x \in \mathbb{R} : x^2 - 2x + 1 = 0\}.$$

Thus, in this case, the predicate is the equation.

Question 1.3.2. What is the predicate in the solution set of the approximation problem in the above example?

By expressing problems as solution sets we reduce the notion of finding a solution to that of checking membership in a set. This is, given a value of the predicate's variable (an input), we must decide whether the predicate is true or false.

1.4 Strings and Formal Languages

We translate problems, or rather their solutions sets, to sets of *strings* to make them processable by a computer device. So, strings are the basic data structure in theory of computation. This subsection outlines the main features of strings. Let's start by making an important clarification,

Definition 1.4.1. By a *symbol*, we understand a single, atomic character, that cannot be expressed as the concatenation of other symbols.

Example 1.4.1. According to this definition,

1. 0, 1, 3, and 5 are all symbols in the numerical system, but 10, 315, and 1035, are not because they are concatenations of symbols;
2. For the same reason, *a*, *b*, *c* are symbols in the latin alphabet, but *cab* and *abcab*, are not.

Base on this concept of symbol, we define

Definition 1.4.2. An alphabet is a finite set of symbols.

Following a long established tradition we denote alphabets with a capital Σ. Thus,

Example 1.4.2. The following sets are alphabets:

1. $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$ is an alphabet called *decimal alphabet*.
2. $\Sigma = \{0,1\}$ is an alphabet called *binary alphabet*.

3. $\Sigma = \{a, b, c, d, ..., x, y, z\}$ is an alphabet called *latin alphabet*.

Remark 1.4.1. It is worth remaking that alphabet cannot be the empty set. Also notice that the union and the non-empty intersection of alphabets is also an alphabet.

Definition 1.4.3. A string w over an alphabet Σ is a finite concatenation of elements of Σ. The length of a string, denoted $|w|$, is the number of concatenations, counting all repetitions of symbols.

Example 1.4.3.

1. $w = 83458335258$ is a string of length $|w| = 11$ over the decimal alphabet.
2. $w = 1101001$ is a string of length $|w| = 7$ over the binary alphabet.
3. $w = hello$ is a string of length $|w| = 5$ over the latin alphabet.

1.4.1 String Operations

No information processing is possible without the ability to read and write strings. Given a string $w = w_1 w_2 \cdots w_n$, formed with the symbols $w_i \in \Sigma$, $i = 1, ..., n$, the operation of *read* and *write* can be modeled as a function and its inverse, as follows:

$$\rho(w_1 w_2 \cdots w_n) = (w_1, w_2, ..., w_n), \tag{1.1}$$
$$\omega(w_1, w_2, \cdots, w_n) = w_1 w_2 \cdots w_n. \tag{1.2}$$

When reading, we identify each symbol in the string and the order in which they appear. Both aspects, symbol and order, convey the meaning of the string. Function (1.1) implements this model of reading by identifying each symbol and placing it in the appropriate order in an n-tuple. In turn, when writing, we select the symbols to compose a word and the order in which they appear. Thus, writing can be modeled as the inverse of reading, which is mapping (1.2).

Example 1.4.4.

$$\rho(hello) = (h, e, l, l, o) \tag{1.3}$$
$$\omega(h, e, l, l, o) = hello. \tag{1.4}$$

Definition 1.4.4. The concatenation of a string w and a string z is the string $u = wz$.

Example 1.4.5.

$$w = dog \wedge z = house \to u = wz = doghouse. \tag{1.5}$$

Remark 1.4.2. Notice that $|wz| = |w| + |z|$.

Definition 1.4.5. The *null string*, also called *empty string*, denoted by ε, is the neutral element for the operation of string concatenation. This is, for any given string w,

$$w = w\varepsilon = \varepsilon w.$$

By convention, $|\varepsilon| = 0$, and is the only string of length zero.

String concatenation is also associative, this is,

$$\forall w, y, z \, \text{strings over } \Sigma, (wy)z = w(yz);$$

but is not commutative, as for instance,

$$doghouse \neq housedog.$$

1.4.2 An Extension of the Concept of Alphabet

In the previous subsection we defined *alphabet* as a finite set of symbols. But sometimes in practice, finite sets of strings are used as alphabets, as well. This is for instance the case of ASCII, where each letter in the latin alphabet and each punctuation symbol is represented by a different binary string. In order to account for this potential variant, we introduce

Definition 1.4.6. A *string alphabet* is a finite set of strings on which the read mapping is well-defined.

Example 1.4.6. The set $\Sigma = \{00, 01\}$ is a string alphabet, as each string over Σ has a unique representation as a tuple having 00 or 01 in its components. The set $\Sigma = \{0, 00, 01, 001\}$ is not a string alphabet as, for instance, the string 0001 could be read as $(00, 01)$ or $(0, 0, 01)$, among other possibilities.

An ambiguous read operation impedes the processing of the string.

Remark 1.4.3. Notice that the union of two string alphabets is not necessarily a string alphabet.

Question 1.4.1. Can you find examples of unions of string alphabets that are not string alphabets?

Question 1.4.2. Alice said that any finite union of strings of the same length is a string alphabet. Is Alice right?

1.4.3 Formal Languages

A word of caution, in theory of computation formal languages are not intended to model computer languages.

Definition 1.4.7. A formal language L, is a set of strings over an alphabet.

By default, an alphabet Σ, and the empty set $\emptyset$ are languages. The basic operations with formal languages are defined next.

Definition 1.4.8. The operations with languages are:

1. All *set operations*.

2. *Language concatenation*: Given L_1 and L_2 languages over the same alphabet Σ, their concatenation is

$$L = L_1 L_2 = \{w : w = yz, y \in L_1 \wedge z \in L_2\}. \tag{1.6}$$

3. *Kleene star operation*: Given a language L, the Kleene star of L, denoted L^*, is the language that results from the following inductive construction:

$$L^0 = \{\varepsilon\}, \tag{1.7}$$
$$L^1 = L, \tag{1.8}$$
$$L^j = L^{j-1}L;\ j \geq 2. \tag{1.9}$$

Then,

$$L^* = \bigcup_{j=0}^{\infty} L^j. \tag{1.10}$$

Let's prove first the distribution of concatenation over unions.

Theorem 1.1 (Distribution of Concatenation over Unions)**.** *Let A, B and C be languages over the same alphabet. Then,*

1. $A(B \cup C) = AB \cup AC$,

2. $(A \cup B)C = AC \cup BC$

Proof. We prove only statement 1. The proof of 2 is similar.

$$\begin{aligned} A(B \cup C) &= \{w : w = vz,\ v \in A \wedge z \in B \cup C\} \\ &= \{w : w = vz,\ v \in A \wedge (z \in B \vee z \in C)\} \\ &= \{w : w = vz,\ (v \in A \wedge z \in B) \vee (v \in A \wedge z \in C)\} \\ &= AB \cup AC. \end{aligned}$$

□

Let's illustrate now the Kleene star operation.

Example 1.4.7. Let $L = \{a, b\}$. Then,

$$\begin{aligned}
L^0 &= \{\varepsilon\} \\
L^1 &= \{a, b\} \\
L^2 &= \{a, b\}\{a, b\} = \{aa, ab, ba, bb\} \\
L^3 &= \{aa, ab, ba, bb\}\{a, b\} = \{aaa, aba, baa, bba, aab, abb, bab, bbb\} \\
&\vdots \\
L^n &= \{w : w \,\text{string over}\, \{a, b\}, \, |w| = n\}.
\end{aligned}$$

Thus,

$$L^* = \{\varepsilon\} \cup \{w : \text{string over}\, \{a, b\}\}.$$

Lemma 1.1. *Let Σ be an alphabet. Then,*

1. $\forall i \in \mathbb{N}, \Sigma^i \cap \Sigma^{i+1} = \emptyset$*; and*
2. $\forall i \in \mathbb{N}, |\Sigma^i| = |\Sigma|^i$.

Proof. We prove each statement separately.
Proof of 1. We first demonstrate by induction that $\forall a \in \Sigma^n, |a| = n$.
Case $n = 1$. Since Σ is an alphabet, each $a \in \Sigma$ is a symbol, and therefore, $|a| = 1$.
Inductive hypothesis: $\forall w \in \Sigma^n, |w| = n$ is true for n. Then,

$$\begin{aligned}
\Sigma^{n+1} &= \Sigma^n \Sigma \\
&= \{wa : w \in \Sigma^n \wedge a \in \Sigma\},
\end{aligned}$$

Since $|wa| = |w| + |a|$, using the inductive hypothesis we get $|wa| = n + 1$.

As a consequence of the statement we have just proved, the elements in Σ^i cannot be in Σ^{i+1}, and viceversa. Therefore, $\forall i \in \mathbb{N}, \Sigma^i \cap \Sigma^{i+1} = \emptyset$.
Proof of 2. By induction.
Case $n = 1$ holds because $\Sigma^1 = \Sigma$.
Inductive hypothesis: $|\Sigma^n| = |\Sigma|^n$, for some n. Then,

$$\begin{aligned}
|\Sigma^{n+1}| &= |\Sigma^n \Sigma| \\
&= |\Sigma^n||\Sigma|.
\end{aligned}$$

The last step holds because of the definition of language concatenation and the first statement in this lemma (which was already proved). Now, using the inductive hypothesis:

$$|\Sigma^n||\Sigma| = |\Sigma|^n |\Sigma| = |\Sigma|^{n+1}.$$

□

Question 1.4.3. Is it the case that for each $i \in \mathbb{N}$, $L^i \cap L^{i+1} = \emptyset$? If this is indeed the case, prove it. Otherwise, provide a counterexample.

Corollary 1.1. *Let Σ be an alphabet. Then, for each $n \in \mathbb{N}$, $n \geq 1$,*

$$|\bigcup_{i=0}^{n} \Sigma^i| = \begin{cases} \frac{n(n+1)}{2}, & \text{if } |\Sigma| = 1 \\ \frac{|\Sigma|^{n+1}-1}{|\Sigma|-1}, & \text{if } |\Sigma| > 1. \end{cases}$$

Recall that

Definition 1.4.9. A set S is *countable* if there is a one-to-one and onto map $f : S \to R$, with $R \subset \mathbb{N}$. If R is infinite, S is called *countably infinite*.

Since it has been proved that the countable union of finite sets is a countable set, we get yet another important corollary:

Corollary 1.2. *Let Σ is an alphabet, then, Σ^* is countably infinite.*

Proposition 1.1. *Let Σ be an alphabet. Then, $\forall L$ language over Σ, $L \subset \Sigma^*$.*

Proof. Since L is a language over Σ, for each $w \in L$, there is an $n > 0$ such that, $w = w_1 \cdots w_n$, where $w_i \in \Sigma$, $i = 1, ..., n$ is a symbol. Therefore,

$$(\forall w \in L)\, \exists n \in \mathbb{N}, w \in \Sigma^n.$$

□

Consequently,

$$L \subset \bigcup_{n=0}^{\infty} \Sigma^n = \Sigma^*.$$

Theorem 1.2 (Properties of Kleene Star Operation)**.** *Let A and B be languages over the same alphabet. Then,*

1. $\{\varepsilon\}^* = \phi^* = \{\varepsilon\}$,
2. *If $A \subset B$, then $A^* \subset B^*$,*
3. $(A^*)^* = A^*$,
4. $(A \cup B)^* = (A^* \cup B^*)^* = (A^* B^*)^*$.

Proof. Consider each statement separately.

1. *Proof is left as an exercise.*
2. Since $A \subset B$, $\forall x \in A, x \in B$. Thus, each concatenation of elements in A is also a concatenation of elements in B. Consequently, $A^* \subset B^*$.

3. To show that *(a)* $A^* \subset (A^*)^*$ and *(b)* $(A^*)^* \subset A^*$.

   Proof of (a). Since $A \subset A^*$, we get $A^* \subset (A^*)^*$ as a consequence of 2.

   Proof of (b). If $x \in (A^*)^*$, then

   $$(\exists n \in \mathbb{N})\, x = x_1 x_2 \cdots x_n \wedge x_i \in A^*.$$

   Now,

   $$(\forall i = 1, ..., n)(\exists k_i \in \mathbb{N})\, x_i = x_i^{(1)} \cdots x_i^{(k_i)} \wedge (\forall j = 1, ..., k_i)\, x_i^{(j)} \in A.$$

   Therefore, $x = x_1^{(1)} \cdots x_1^{(k_1)} x_2^{(1)} \cdots x_2^{(k_2)} \cdots x_n^{(1)} \cdots x_n^{(k_n)}$ which is a concatenation of elements in A. Therefore, $(A^*)^* \subset A^*$.

4. We demonstrate first that $(A \cup B)^* = (A^* \cup B^*)^*$

   Proof of $(A \cup B)^* \subset (A^* \cup B^*)^*$. Since

   $$A \subset A^*$$
   $$B \subset B^*.$$

   Thus, $(A \cup B) \subset (A^* \cup B^*)$. By applying 2,

   $$(A \cup B)^* \subset (A^* \cup B^*)^*.$$

   Proof of $(A^* \cup B^*)^* \subset (A \cup B)^*$. Since $A \subset A \cup B$ and $B \subset A \cup B$, by applying 1 we get

   $$A^* \subset (A \cup B)^*, \text{ and}$$
   $$B^* \subset (A \cup B)^*.$$

   Thus, $A^* \cup B^* \subset (A \cup B)^*$. By applying 2,

   $$(A^* \cup B^*)^* \subset ((A \cup B)^*)^* = (A \cup B)^*.$$

   We demonstrate now that $(A^* B^*)^* = (A \cup B)^*$

   Proof of $(A \cup B)^* \subset (A^* B^*)^*$. Since

   $$A \subset A^* B^*$$
   $$B \subset A^* B^*,$$

   $(A \cup B) \subset A^* B^* \cup A^* B^* = A^* B^*$. By applying 2,

   $$(A \cup B)^* \subset (A^* B^*)^*.$$

   Proof of $(A^* B^*)^* \subset (A \cup B)^*$. Since

   $$A^* \subset A^* \cup B^*$$
   $$B^* \subset A^* \cup B^*,$$

it follows that

$$A^*B^* \subset (A^* \cup B^*)(A^* \cup B^*).$$

Therefore,

$$(A^*B^*)^* \subset (A^* \cup B^*)^* = (A \cup B)^*.$$

□

1.4.4 Example of Formal Language Encoding of a Problem

Let consider the simple problem of deciding whether a natural number is even or not. In set-theoretical terms, the problem is:

$$EVEN = \{n \in \mathbb{N} : (\exists k \in \mathbb{N})\, n = 2k\}.$$

We use a usual encoding of natural numbers (unsigned integers) over $\Sigma = \{0,1\}$. This is, we represent a number with the string over Σ formed with the coefficients of its expansion on base 2. By doing this, we get the formal language

$$EVEN = \{w \in \Sigma^* : w \text{ ends in } 0\}.$$

The predicate in the formal language representation of $EVEN$ is correct as a number is even if and only if its expansion on base 2 ends in 0.

1.5 General Notion of Automaton

In general, the concept of automation conveys the idea of self-operation. While full automation is achieved in the natural world in living organisms, humankind has produced devices that imitate, or create the illusion of automation. An instance of such illusion is an automatic door, that opens under the action of an actuator that senses the proximity of a person. Computers are another example of artificial automata.

The automatic behavior of a computing device is created by a finite sequence of instructions. In theory of computation, the model of computer automation concentrates on the sequence of instructions that induces its automatic behavior. The basic instruction being *read* a symbol from a string and *write* a symbol on a string. Computing devices execute these instructions in a discrete sequence of pulses, which are modeled in theory of computation as states of the automaton.

Thus, an automaton is a mathematical entity that reads or writes on a string, character by character, and changes its state with each reading or writing. And, as discussed earlier, the aim of an automaton is to solve a decision problem by deciding whether a given input string belongs to the solution set of the problem. The automaton does this by *accepting* or *rejecting* the input string.

Chapter 2

Regular Languages

This chapter introduces the concept of regular language, and the mechanisms for its generation with formal language operations, and their recognition by parsing. Then, the main results in the theory of finite state automaton, which is the model of the automated solution of decision problems whose solution is encoded as a regular language, are discussed.

2.1 Regular Languages and Regular Expressions

2.1.1 Construction of Regular Languages

We start with an *inductive* definition of *regular language*.

Definition 2.1.1. Given an alphabet Σ, we declare

1. $\{a\}$, $\forall a \in \Sigma$; $\emptyset$, and $\{\varepsilon\}$ to be regular languages;
2. If A and B are regular languages, so is $A \cup B$;
3. If A and B are regular languages, so is AB; and
4. If A is a regular language, so is A^*.

The language operations of *union, concatenation,* and *Kleene star* are called **regular operations.**

Example 2.1.1. The simplest example of regular language over an alphabet Σ is a subset of $A \subset \Sigma$, including Σ itself. Indeed, $A \subset \Sigma$ is the finite union:

$$A = \bigcup_{i=1}^{n} \{a_i\},$$

where $a_i \in A$, $i = 1, \ldots, n$. Because of 1 in Definition 2.1.1 $\{a_i\}$ is a regular language for each $i = 1, \ldots, n$; and because of 2 in Definition 2.1.1, A is a regular language.

Similarly, Σ^* is regular, because is the Kleene star of a (*we've just prove it!*) regular language.

Proposition 2.1. *If L is a finite language, then L is regular.*

Proof. If L is finite over an alphabet Σ, then $L = \{w_1, w_2, ..., w_m\}$, with $w_i \in \Sigma^*$, $i = 1, ..., m$. Thus,

$$(\forall i = 1, ..., m)(\exists k_i \in \mathbb{N}) w_i = w_i^{(1)} \cdots w_i^{(k_i)};$$

where for each $1 \leq j \leq k_i$, $w_i^{(j)} \in \Sigma$. By expressing each string in L, and each element in Σ as a language with one word, we get

$$\{w_i\} = \{w_i^{(1)}\}\{w_i^{(2)}\} \cdots \{w_i^{(k_i)}\}.$$

Since each $w_i^{(k_i)} \in \Sigma$, $\{w_i^{(k_i)}\}$ is a regular language. Thus, $\{w_i\}$ is a finite concatenation of regular languages, and therefore, a regular language. This means that L is a finite union of regular languages, and therefore, a regular language. □

Example 2.1.2. Let's consider some cases of infinite languages defined with predicates.

1. Is $L = \{w \in \{0,1\}^* : w \text{ starts and ends with } 0\}$ a regular language?

   Answering this question requires expressing the predicate in terms of regular operations of regular languages. In this particular case, we have:

   $$L = \{0\}\{0,1\}^*\{0\},$$

   which is the concatenation of three regular languages. Therefore, L is regular.

2. Is $L = \{w \in \{0,1\}^* : |w| \text{ is a multiple of } 3 \vee |w| \text{ is a multiple of } 5\}$ a regular language?

   Again, we look for an expression of L as regular operations on regular languages. In this case we have

   $$L = (\{0,1\}\{0,1\}\{0,1\})^* \cup (\{0,1\}\{0,1\}\{0,1\}\{0,1\}\{0,1\})^*.$$

3. $EVEN = \{0,1\}^*\{0\}$ is also a regular language.

4. A *palindrome* is a string which reads the same backward as forward, such as *madam*, *radar* or 0110. Let $L = \{w \in \{0,1\} : w \text{ is a palindrome}\}$. We see that there is no clear way to express L in terms of regular operations over regular languages. Indeed, we will show by the end of this chapter that this is not a regular language.

Expressing a regular language in terms of regular regular operations over regular languages is simplified by the use of regular expressions. Next is a inductive definition of regular expression.

Definition 2.1.2. A *regular expression* over an alphabet Σ is a string over the alphabet $\Sigma \cup \{^*, \cup, (,)\}$ formed with the following rules:

1. $(\forall a \in \Sigma)$ a is a regular expression;
2. ε, and $\emptyset$ are regular expressions;
3. $(R_1 \cup R_2)$ is a regular expression if R_1 and R_2 are regular expressions;
4. $(R_1 R_2)$ is a regular expression if R_1 and R_2 are regular expressions; and
5. R^* is a regular expression if R is a regular expression.

Remark 2.1.1. It is important to keep in mind that, despite similarities with expressions of regular languages as regular operations over regular languages, a regular expression is a string and therefore, it contains no language. In a regular expression,

1. a with $a \in \Sigma$, represents the language $\{a\}$;
2. ε represents the language $\{\varepsilon\}$, and $\emptyset$ the empty language;
3. $R_1 \cup R_2$ and $R_1 R_2$ represent the union and concatenation of the languages represented by R_1 and R_2, respectively;
4. R^* represents the language $\{\varepsilon, R, RR, RRR, ...\}$.

Example 2.1.3. A regular expression representing language $L = \{0\}\{0,1\}^*\{0\}$ is

$$R = 0(0 \cup 1)^* 0.$$

The existence of a string representation for an infinite language is a unique feature of regular languages and it has several important consequences. Among them is the verification of its correctness through the parsing of the regular expression. Each parse tree of a regular expression has the regular expression as root, its internal nodes are the regular operations, which appear in a order consistent with the construction of the expression from the leaves up to the root. The leaves of the parse tree are the symbols in the alphabet that are included in the regular expression. Figure 2.1.1 is the parse tree of the regular expression $R = 0(0 \cup 1)^* 0$ of the language $L = \{0\}\{0,1\}^*\{0\}$.

Remark 2.1.2. It is important to emphasize that a regular expression contains no other symbols than those in the alphabet $\Sigma \cup \{^*, \cup, (,)\}$. In particular, neither the symbol Σ used to represent the alphabet itself, nor expressions of the form a^4 or a^n can be made part of a regular expression. Expressions such as

$$\Sigma\Sigma, \Sigma^*, 0^4,$$

are to be taken as short cuts used in natural language but not as regular expressions.

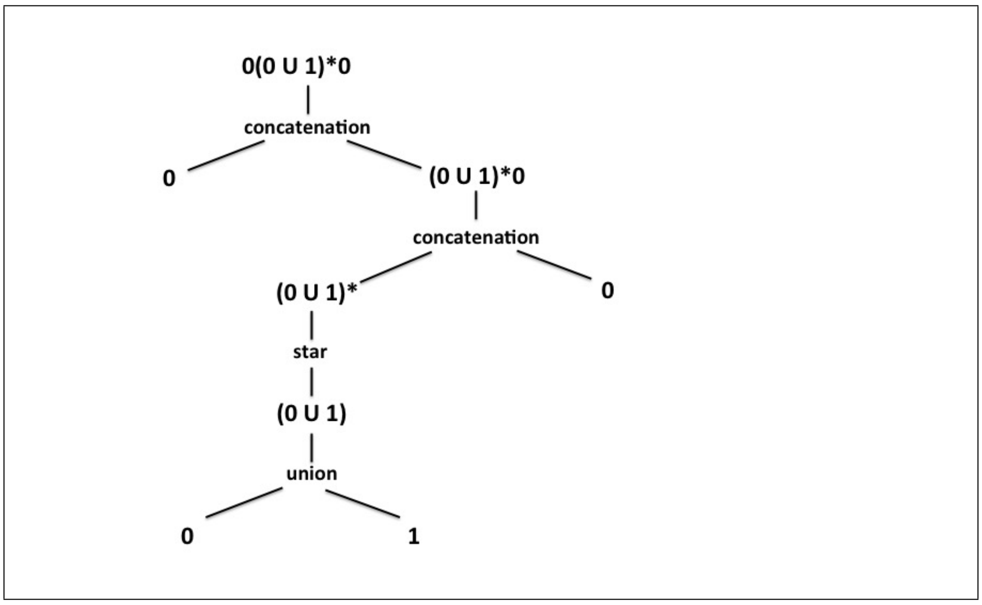

Figure 1: Parse tree of regular expression $0(0\cup 1)^*0$

Regular expressions get easily very large. In some instances, they can be greatly reduced and organized using some simple algebraic relations, like the ones in the next theorem and corollary.

Theorem 2.1 (Properties of Regular Expressions)**.** *Let R_1, R_2 and R_3 be regular expressions. Then,*

1. $R_1(R_2\cup R_3) = R_1R_2\cup R_1R_3$ *and* $(R_1\cup R_2)R_3 = R_1R_3\cup R_2R_3$*;*

2. $(R_1^*)^* = R_1^*$*; and*

3. $(R_1\cup R_2)^* = (R_1^*\cup R_2^*)^* = (R_1^*R_2^*)^*$.

Proof. By identifying each R_1, R_2 and R_3 with regular languages A, B and C, property 1 follows directly from Theorem 1.1, and properties 2 and 3 from Theorem 1.2, 3 and 4; respectively. □

Remark 2.1.3. By analogy with the construction of the sets L^n in the definition of L^*, we denote for a symbol $a\in\Sigma$,

$$\begin{aligned} &a^0 = \varepsilon\\ &a^1 = a\\ &a^j = a^{j-1}a, (\forall j\in\mathbb{N}, j\geq 0). \end{aligned}$$

Thus, we may eventually argue that $a^* = \varepsilon\cup a^1\cup\ldots a^n\ldots$ provided that the right-hand side of this equation is not considered a regular expression.

Corollary 2.1 (Conjunctive Normal Form of Regular Expressions)**.** *Each regular expression R can be written as*

$$R = R_1 \cup R_2 \cup \cdots \cup R_m; \tag{2.1}$$

where each R_i contains no unions.

Proof. The conjunctive normal form of R is produced just by applying Theorem 2.1, 1 or 3, either to eliminate unions affected by concatenations or unions under Kleene star operations. $\square$

Example 2.1.4. Let $\Sigma = \{a,b,c\}$. Then, the regular expression $c(a\cup ab)\cup(a\cup ab)^*$ is not in conjunctive normal form. By applying Theorem 2.1 we get

$$\begin{aligned} c(a\cup ab)\cup(a\cup ab)^* &= (ca\cup cab)\cup(a\cup ab)^* \\ &= (ca\cup cab)\cup(a^*(ab)^*)^* \\ &= ca\cup cab\cup(a^*(ab)^*)^*, \end{aligned}$$

which is in conjunctive normal form.

Remark 2.1.4. Another interesting aspect of a regular expression in conjunctive normal form is the separation of the *finite* and *infinite* parts of a regular language, as the infinite segments, meaning segments with an infinite number of strings, are only those that are represented by regular expressions under the Kleene star. In Example 2.1.4 this segment corresponds to $(a^*(ab)^*)^*$

2.2 Finite State Automata

A regular language L is a kind of formal language and as such, as discussed in Chapter 1, we regard it as the encoding of the solution set of a problem. In this section we turn into the computation and verification, this is the *recognition*, of a solution to that problem. What we look for is an abstract structure representing a process able to decide whether a given string $w \in \Sigma^*$ is a member of L.

2.2.1 Definition of Finite State Automata

We can briefly anticipate some characteristics of this structure, before defining it formally. The fact that a regular language L can be represented in a regular expression R, which is a string, indicates that the question of whether another string $w \in \Sigma^*$ is in L can be answered with a sort of *string matching algorithm.* In particular, no extra memory is necessary, as all we need is to keep w and R, and compare them symbol by symbol up until w is matched, or eventually, pass through R without finding a match. The next structure implements this matching algorithm.

Definition 2.2.1. A **finite state automata** is a quintuple $A = (Q,\Sigma,\delta,q_0,F)$ where

1. Q is a finite set of symbols representing the *states of the automaton*;
2. Σ is the alphabet of the language (problem) for which the automaton is designed (programmed);
3. $\delta : Q \times \Sigma \to Q$ is the transition relation, which associates to a state and a symbol (q,a), a transition to a new state $p = \delta(q,a)$; and
4. $F \subset Q$, is the set of *accept states.*

Example 2.2.1. Let $E = (\{a,b\},\{0,1\},\delta,a,\{a\})$, where δ is defined as:

δ	0	1
a	a	b
b	a	b

Table 4: Definition of δ

Remark 2.2.1. Two important remarks:

1. It is worth noticing that *finite state automaton* is a misnomer as all the other structures that serve similar roles but for different kinds of formal languages, are finite state automata, as well. We distinguish among them by using capital letters when referring to this particular finite state automaton.
2. It is customary to represent a finite state automaton with a directed graph. This is convenient only for small automata. Figure 2 is a directed graph representation of the automaton defined in Example 2.2.1. In these graphs, vertices represent states and edges, which are labeled with symbols in the alphabet, represent transitions. Accept states (this is, states in F) are distinguished with a double circle.

2.2.2 Computation with a Finite State Automaton

Definition 2.2.2. A **deterministic Finite State Automata** (DFSA) $A = (Q,\Sigma,\delta,q_0,F)$ processes an input string $w = w_1w_2\cdots w_n$ by executing the next sequence of transitions:

$$\begin{aligned} q_1 &= \delta(q_0,w_1) \\ q_2 &= \delta(q_1,w_2) \\ &\vdots \\ q_n &= \delta(q_{n-1},w_n). \end{aligned}$$

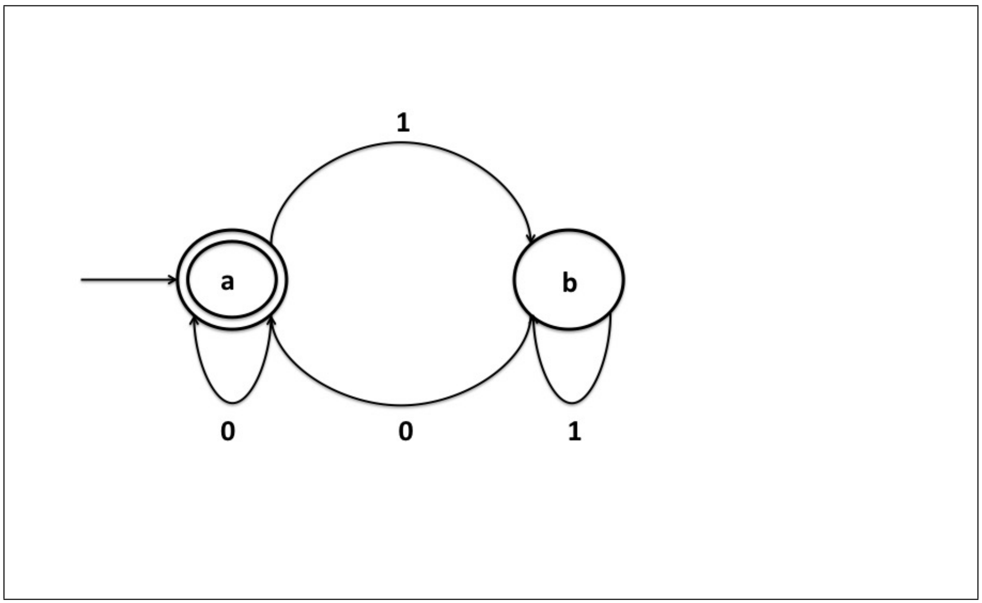

Figure 2: Digraph representation of automaton E

The string $q_0 q_1 \cdots q_n \in Q^*$ is called **trace** of the computation. A string w is **accepted** if and only if $q_n \in F$. The set

$$L(A) = \{w \in \Sigma^* : A \text{ accepts } w\}$$

is said to be the **language** or A.
A language L over Σ is said to be **recognized** by A if and only if $L = L(A)$.

Example 2.2.2. Consider the string $1010 \in \{0,1\}^*$, and the automaton E of Definition 2.2.1. The computation of 1010 by E is

$$\begin{aligned} b &= \delta(a,1) \\ a &= \delta(b,0) \\ b &= \delta(a,1) \\ a &= \delta(b,0). \end{aligned}$$

The trace is *ababa*. Since the computation ends in state a, the string 1010 is accepted. Instead, the computation of string 011

$$\begin{aligned} a &= \delta(a,0) \\ b &= \delta(a,1) \\ b &= \delta(b,1). \end{aligned}$$

whose trace is *aabb*, indicates that E rejects the string.

What is the language of E? Since E is a fairly simple automaton, we may answer this question just by inspecting Table 4. A quick look reveals that $\delta(a,0) =$

$\delta(b,0)=a$, are the sole transitions that return a, the accept state of E. It follows that $L(E)=EVEN$, or what is the same, E solves the $EVEN$ problem. In general, finding $L(A)$ for a given automaton A requires an algorithm that is better described after the introduction of Nondeterministic Finite State Automata.

Remark 2.2.2. It is easy to see that a trace of a computation is the result of a combination of three main types of sequences of transitions, namely:

1. Sequential:

   $$\begin{aligned} q_2 &= \delta(q_1,a_1) \\ q_3 &= \delta(q_2,a_2). \end{aligned}$$

   See Figure 3;

2. Branching:

   $$\begin{aligned} q_2 &= \delta(q_1,a_1) \\ q_3 &= \delta(q_1,a_2). \end{aligned}$$

   See Figure 3;

3. Looping:

   $$\begin{aligned} q_2 &= \delta(q_1,a_1) \\ q_3 &= \delta(q_2,a_2) \\ &\vdots \\ q_1 = q_n &= \delta(q_{n-1},a_{n-1}). \end{aligned}$$

   See Figure 4

2.3 Nondeterministic Finite State Automata

Nondeterminism is introduced by allowing multiple transitions for a character read, transitions without reading also called ε-transitions, or no transition at all.

Definition 2.3.1. A nondeterministic Finite State Automaton (NFSA), is a 5-tuple $(Q,\Sigma_\varepsilon,\delta,q_0,F)$ where Q, q_0 and F are just as in the definition of DFSA, but

$$\begin{aligned} &\Sigma_\varepsilon = \Sigma\cup\{\varepsilon\}, \text{ and} \\ &\delta : Q\times\Sigma_\varepsilon \to \mathscr{P}(Q) \end{aligned}$$

where $\mathscr{P}(S)$ is the set of all subsets of a set S.

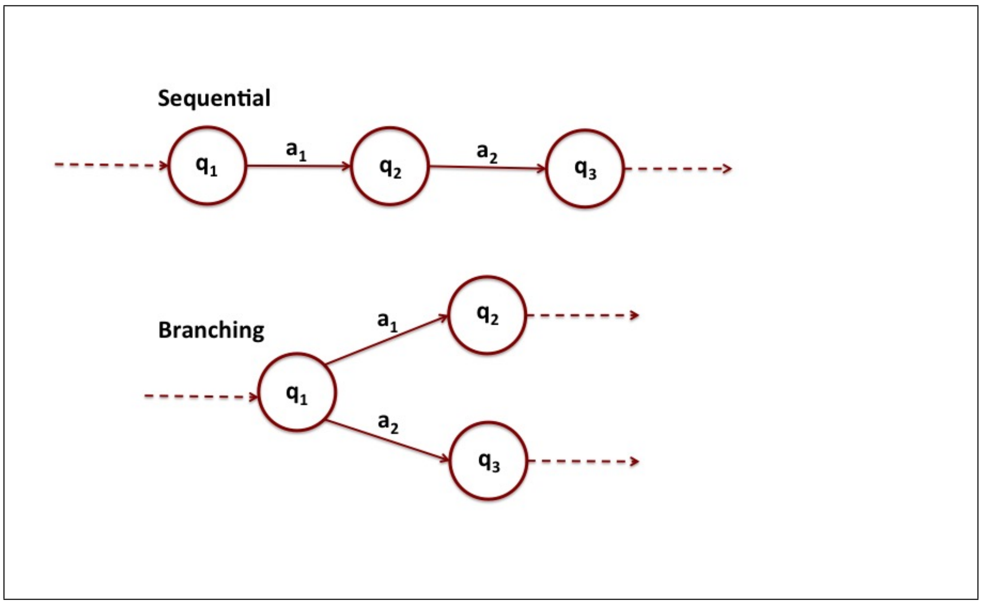

Figure 3: Sequential and branching transition sequences.

Thus, the transition may take $a \in \Sigma \cup \{\varepsilon\}$ to a set of states $\{q_1, ..., q_n\} \subset Q$,

$$\delta(q,a) = \{a,b,...,x\},$$

or eventually, to no state at all

$$\delta(q,a) = \emptyset.$$

If $a = \varepsilon$, none of the above transitions correspond to the reading of a symbol in the input string. They are *spontaneous transitions*.

Example 2.3.1. We define $N = (Q, \Sigma, \delta, q_0, F)$ as follows:

1. $Q = \{a,b,c\}$
2. $\Sigma = \{0,1,2\}$
3. $\delta : \{a,b,c\} \times \{0,1,2\} \to \mathscr{P}(\{a,b,c\})$ defined by

δ	0	1	2	ε
a	$\{b\}$	$\emptyset$	$\emptyset$	$\emptyset$
b	$\emptyset$	$\{a,c\}$	$\emptyset$	$\{c\}$
c	$\emptyset$	$\emptyset$	$\{a\}$	$\emptyset$

Table 5: Definition of δ

4. $F = \{a\}$

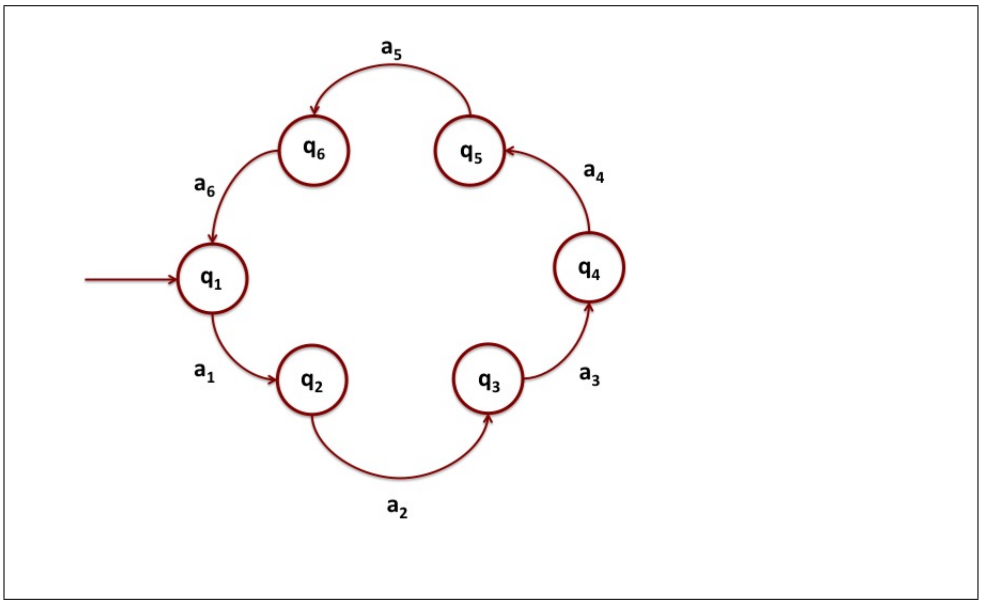

Figure 4: Loop transition sequence.

Figure 5 is a diagraph of this automaton.

Remark 2.3.1. It is worth remarking that a NFSA is not implementable as defined. An NFSA is a theoretical model for solutions obtained through a trial-and-error search, which is expressed with multiple transitions. Transitions to an empty set (*i.e.* no transitions) correspond to trials that fail to return a solution.

Definition 2.3.2. A computation with a NFSA $N = (Q, \Sigma, \delta, q_0, F)$ is a tree rooted at state q_0 whose vertices are states and whose edges represent character reads.

A string w is accepted by a NFSA N if and only if there is a leaf $q \in F$ in the computation tree of w under N.

The set $L(N)$ of all strings accepted by N, is called language of N.

Example 2.3.2. Consider the nondeterministic automaton in Example 2.3.1, and two strings in $\{0,1,2\}^*$, namely $w^{(1)} = 0101$ and $w^{(2)} = 0200$. The computation trees are depicted in Figure 6. The actual depth of each tree increases with the reading of a symbol. Null transitions are represented as segmented branches. Transitions to the empty set are not represented.

According to these results, only string $w^{(1)} = 0101$ is accepted by N, as one of its leaves is a, the accepting state of N.

Remark 2.3.2. As it can be inferred from the notion of computation with a NFSA, a NFSA is not always implementable on a computer device. As is, NFSA computations may generate a number of parallel branches in the computing tree that increases with the length of the input string; eventually surpassing the finite number of logic circuits available in a computer platform. We will address this problem

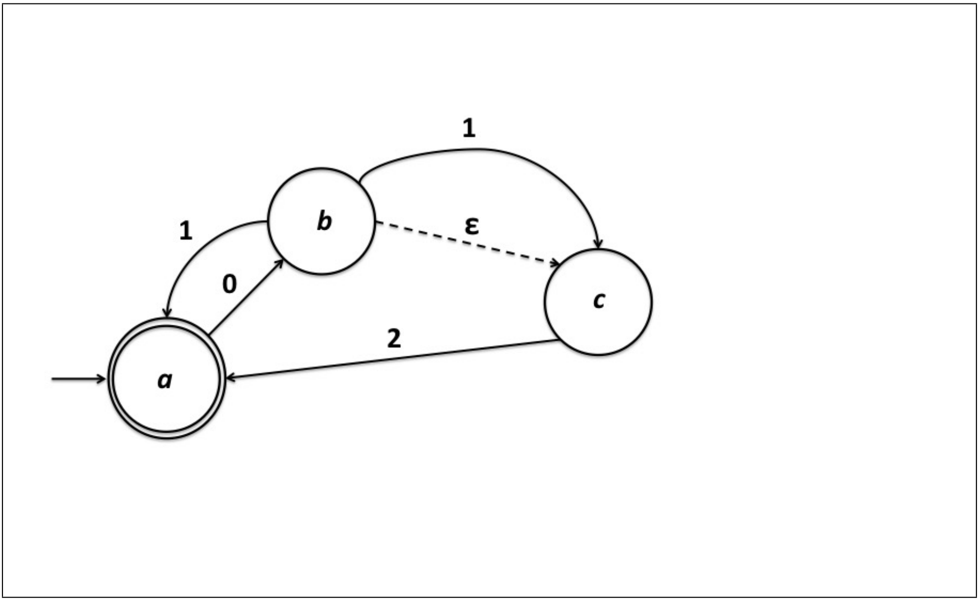

Figure 5: Digraph representation of nondeterministic automaton N

later in this chapter. In the next sections we will use NFSA as a theoretical tool to derive results on DFSA and regular languages.

2.4 The Language of a NFSA

We start by introducing the auxiliary concept of *string transition*.

Definition 2.4.1. By a *string transition* we understand a transition $\delta : Q \times \Sigma^* \to Q$, $\delta(q,w) = r$ with $q,r \in Q$ and $w \in \Sigma^*$.

Theorem 2.2 (The Language of a Finite State Automaton)**.** *If D is a deterministic FSA, then $L(D)$ is a regular language.*

Proof. By construction. We replace $D = (Q,\Sigma,\delta,q_0,F)$ with the NFSA $N = (\bar{Q},\Sigma,\bar{\delta},s,\{a\})$ constructed in the proof of Lemma 2.1, and transform N into a string transition $a = \delta'(s,R)$ where R is a regular expression over the alphabet $\Sigma \cup \{(,),\cup,^*\}$ by

1. Replacing each sequential transition $q_2 = \delta(q_1,a_1); q_3 = \delta(q_2,a_2)$ with a string transition over a_1a_2;
2. Replacing each branching transition $q_2 = \delta(q_1,a_1); q_3 = \delta(q_1,a_2)$ with a string transition over $a_1 \cup a_2$; and
3. Replacing each loop transition $q_2 = \delta(q_1,a_1); q_3 = \delta(q_2,a_2) \cdots q_1 = q_n = \delta(q_{n-1},a_{n-1})$ with a string transition over $(a_1a_2\cdots a_{n-1})^*$

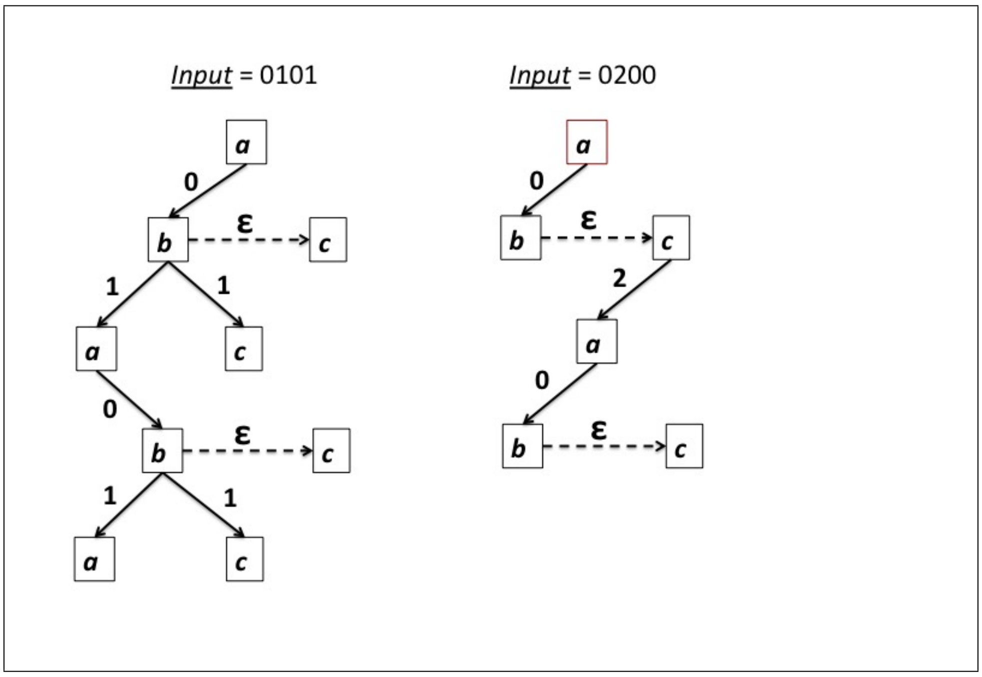

Figure 6: Two computations with automaton N

Then, clearly, R is a regular expression for $L(D)$, and therefore, $L(D)$ is a regular language. □

Example 2.4.1. By applying the three steps listed in the demonstration of theorem 2.2, node by node, we get a regular expression for the language of automaton E in Example 2.2.1.

1. By replacing the loop transition in node b with the corresponding regular expression we get the string transition $\delta'(a,(11^*0)^*) = a$. Figure 7 is the graph of the transformed automaton.

2. And by replacing the two remaining loop transitions in node a by the corresponding regular expression we get the string transition representation of E as $\delta'(s,((11^*0)^* \cup 0^*)^*) = f$. Figure 8 is the graph representation of this transition.

By applying rule 3 in Theorem 2.1 we get

$$((11^*0)^* \cup 0^*)^* = ((11^*0) \cup 0)^*.$$

Let's analyze the resulting regular expression,

$$((11^*0) \cup 0)^* = (\{w \in \{0,1\}^* : w = 1^n0 \vee w = 0 n \in \mathbb{N}\})^*.$$

Clearly, all strings in $(11^*0 \cup 0)^*$ end in 0. Indeed, all strings over $\{0,1\}$ that end in 0 are in $(11^*0 \cup 0)^*$ since each such string can be expressed as a concatenation

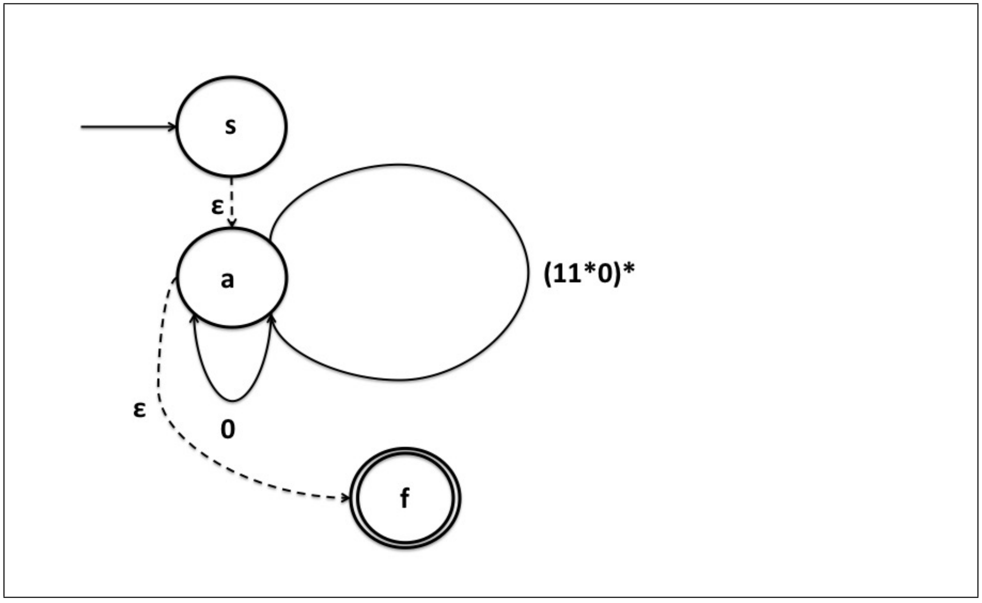

Figure 7: Automaton 15 after replacement of loop transition in node *b* by the corresponding regular expression

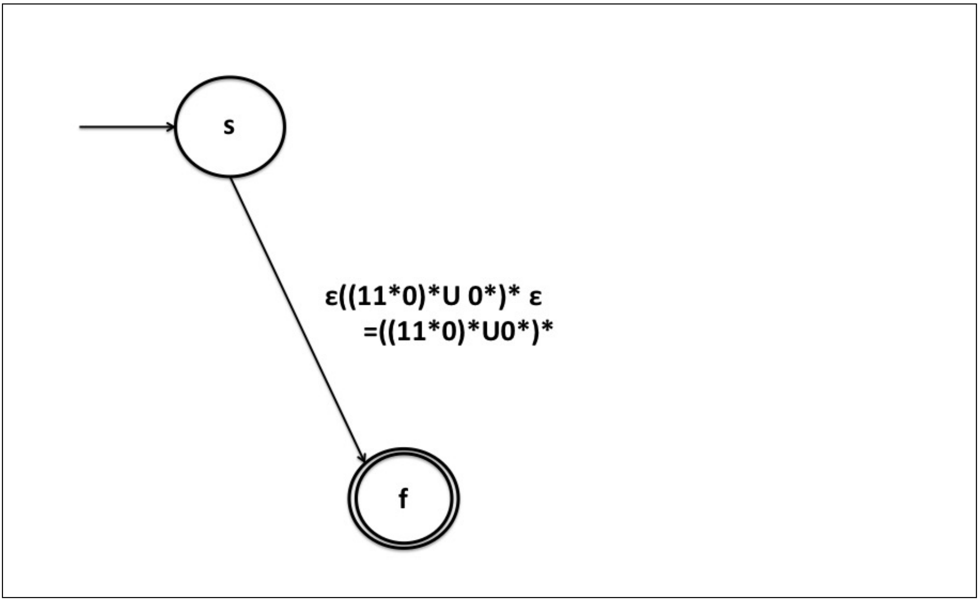

Figure 8: Automaton 15 after replacement of loop transition in node *a* by the corresponding regular expression

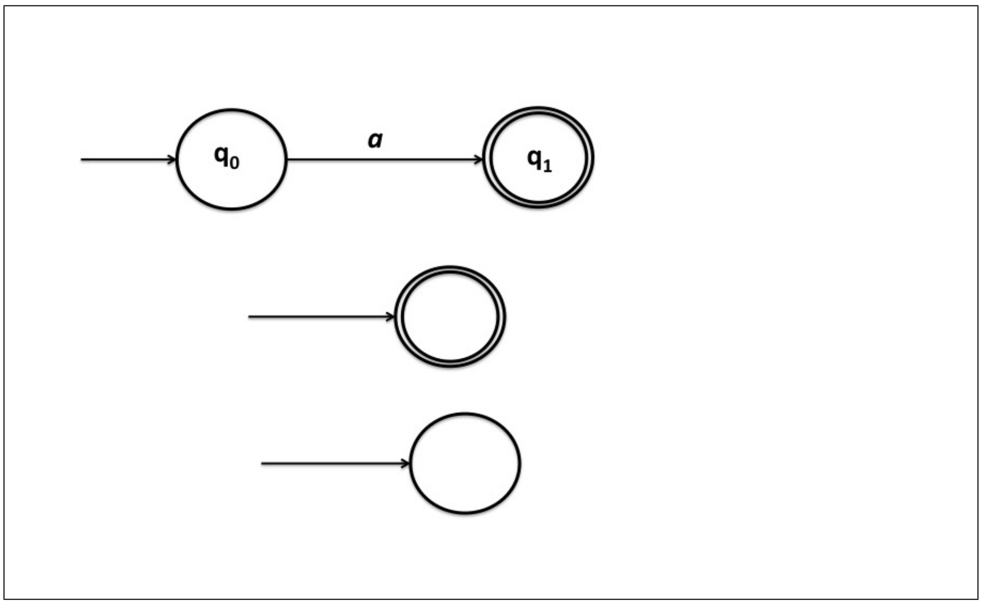

Figure 9: Three basic automata.

of strings of the form $1^{n_1}0, 1^{n_2}0, \ldots, 1^{n_k}0$ where $n_1, n_2, ..., n_k \in \mathbb{N}$, and 0. Thus,

$$(11^*0 \cup 0)^* = \{z \in \{0,1\}^* : z \text{ ends in } 0\}.$$

Therefore, the language of E is the set of all binary strings that end in 0, which turns out to be the solution set of the $EVEN$ problem. Thus, E solves the $EVEN$ problem.

2.5 The Nondeterministic Automaton of a Regular Language

We discuss now a technique for designing a DFSA whose language is the language represented by a given regular expression. The construction of this automaton also uses nondeterministic transitions.

Definition 2.5.1. Given an alphabet Σ, we define the next basic nondeterministic automata:

1. $N_a = (\{q_0, q_1\}, \Sigma, \delta, \{q_1\})$, with $\delta(q_0, a) = q_1$.

2. $N_\varepsilon = (\{q_0\}, \Sigma, \delta, \{q_0\})$, with $\delta(q_0, a) = \emptyset$.

3. $N_\emptyset = (\{q_0\}, \Sigma, \delta, \emptyset)$, with $\delta(q_0, a) = \emptyset$

   Figure 9 depicts each of these automata.

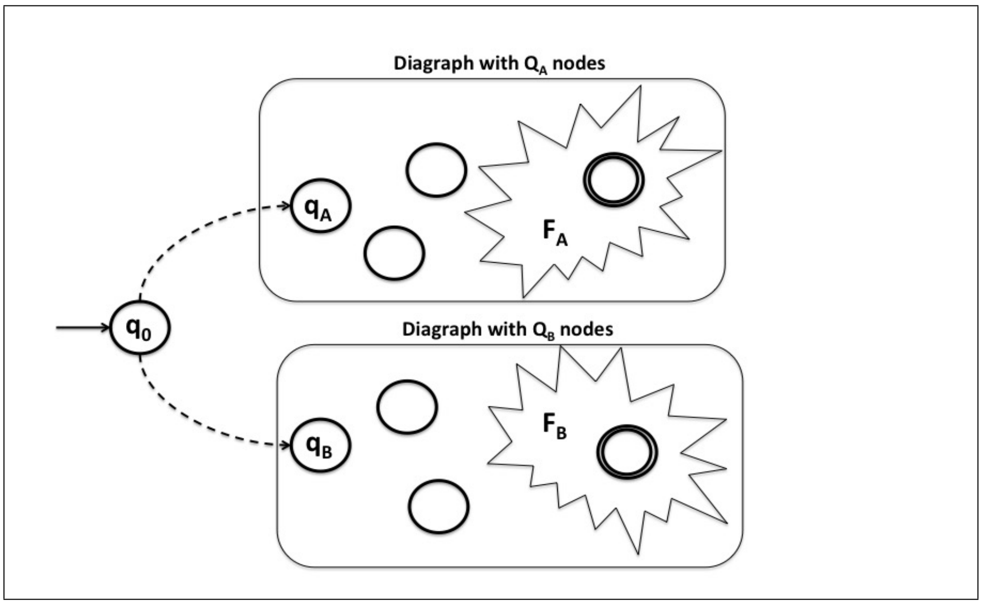

Figure 10: Depiction of automaton $N_{A\cup B}$.

4. Given two automata $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, q_B, F_B$, define $N_{A\cup B} = (Q_A \cup Q_B \cup \{q_0\}, \Sigma, \delta, q_0, F_A \cup F_B)$ with

$$\delta(q,a) = \begin{cases} \{q_A, q_B\}, \text{if } q = q_0 \wedge a = \varepsilon, \\ \delta_A(q,a), \text{if } q \in Q_A, \\ \delta_B(q,a), \text{if } q \in Q_B. \end{cases}$$

   Figure 10 is an abstraction of the diagraph of this automaton.

5. Given two automata $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, q_B, F_B$, define $N_{AB} = (Q_A \cup Q_B, \Sigma, \delta, q_A, F_B)$ with

$$\delta(q,a) = \begin{cases} \delta_A(q,a), \text{if } q \in Q_A \wedge q \notin F_A, \\ \delta_A(q,a), \text{if } q \in F_A \wedge a \neq \varepsilon, \\ \delta_A(q,a) \cup \{q_B\}, \text{if } q \in F_A \wedge a = \varepsilon, \\ \delta_B(q,a) \text{if } q \in Q_B. \end{cases}$$

   Figure 11 is an abstraction of the diagraph of this automaton.

6. Given $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ define $N_{A^*} = (Q_A \cup \{q_0\}\Sigma, \delta, q_A, F_A \cup \{q_0\})$ with

$$\delta(q,a) = \begin{cases} \delta_A(q,a), \text{if } q \in Q_A \wedge q \notin F_A, \\ \delta_A(q,a), \text{if } q \in F_A \wedge a \neq \varepsilon, \\ \delta_A(q,a) \cup \{q_A\}, \text{if } q \in F_A \wedge a = \varepsilon, \\ \{q_A\}, \text{if } q = q_0 \wedge a = \varepsilon. \end{cases}$$

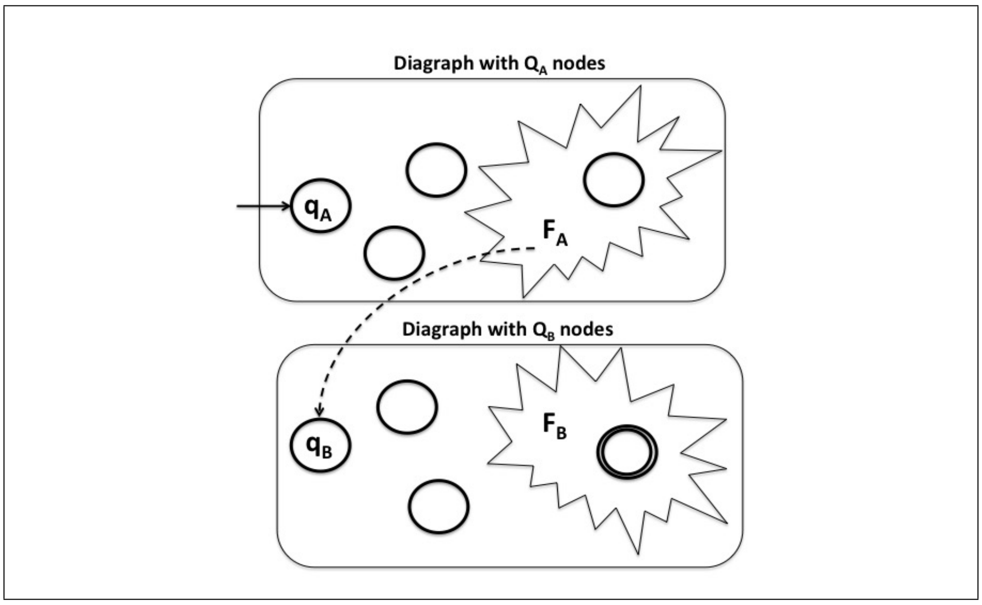

Figure 11: Depiction of automaton N_{AB}.

Figure 12 is an abstraction of the diagraph of this automaton.

Theorem 2.3. *Each regular language is recognized by a NFSA.*

Proof. Let R be a regular expression representing a given regular language. The next procedure constructs an NFSA that recognizes R.

1. Express R in normal conjunctive form, if necessary.
2. Parse R
3. Create an N_a automaton for each $a \in \Sigma$ that is a leaf in the parse tree of R.
4. Construct the automaton bottom up using $N_{A \cup B}$ for unions, N_{AB} for concatenations, and N_{A^*} for Kleene star operations.

□

Example 2.5.1. Here is a simple illustration of the construction of a nondeterministic automaton that recognizes the regular language represented by $01^* \cup 0$.

1. The expression is in conjunctive normal form.
2. Figure 13 is the parse tree of $01^* \cup 0$.
3. The basic automata are represented on top, in Figure 14.
4. Steps in 4 are depicted in the bottom, in Figure 14.

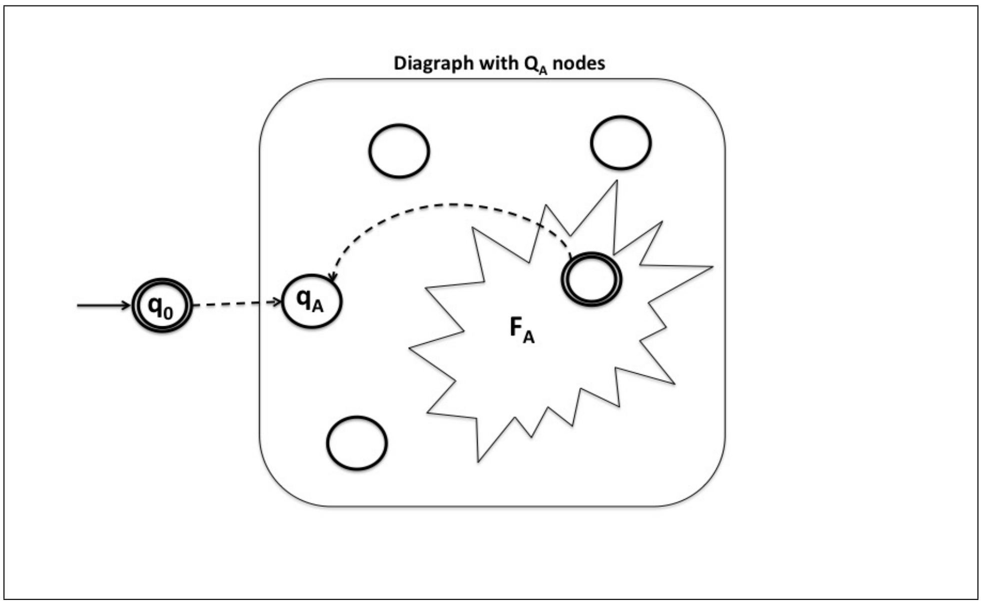

Figure 12: Depiction of automaton N_{A^*}.

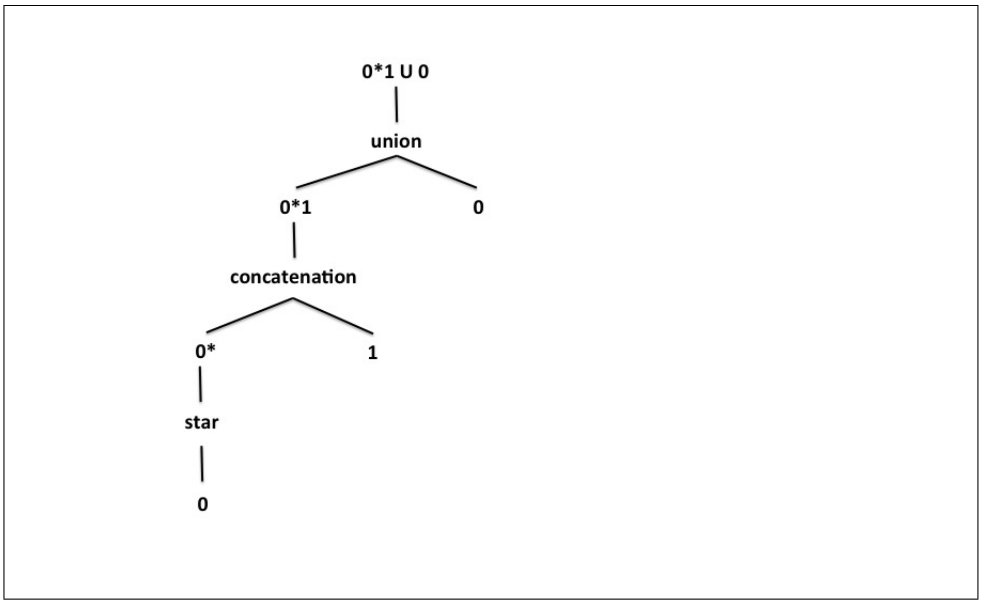

Figure 13: Parse tree for $01^* \cup 0$.

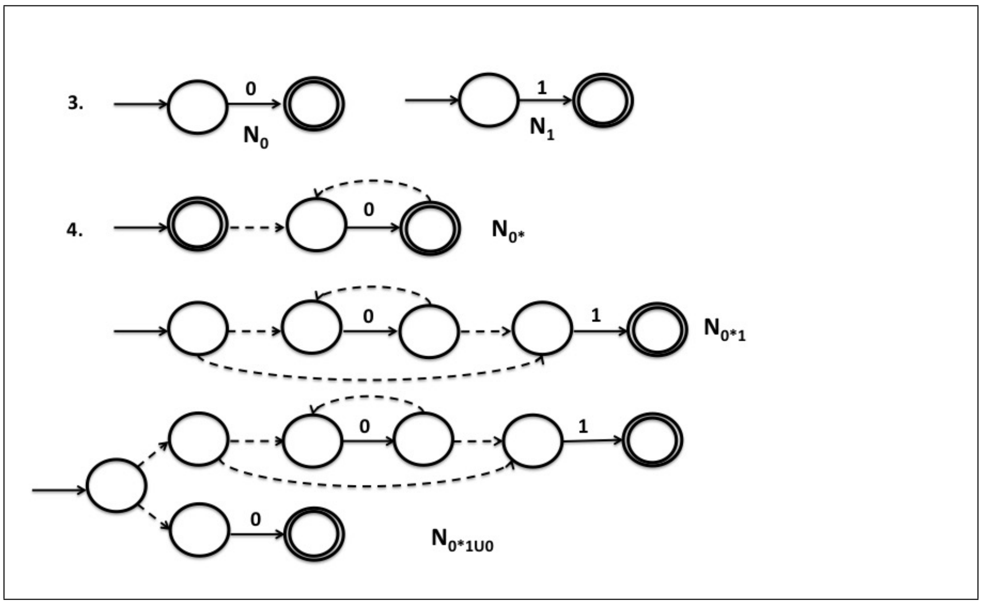

Figure 14: Construction of NFSA for recognizing the language represented by $01^* \cup 0$.

2.6 Equivalence of NFSA and DFSA

Definition 2.6.1. Two finite state automata are said to be equivalent if they recognize the same language.

Remark 2.6.1. Definition 2.6.1 refers not only to Finite State Automata but to all automata with a finite number of states.

Lemma 2.1. *Each DFSA D has an equivalent NFSA N that starts with a null transition and has a single accept state.*

Proof. Proof of existence. Let $D = (Q, \Sigma, \delta, q_0, F)$, and s and a symbols that are not in Q. Let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\bar{Q} = Q \cup \{s, a\}$, $\bar{F} = \{a\}$ and

$$\bar{\delta} : \bar{Q} \times \Sigma_\varepsilon \to \mathscr{P}(\bar{Q});$$

$$\bar{\delta}(q,b) = \begin{cases} \{\delta(q,b)\} & \text{if } q \in Q, b \in \Sigma; \\ \{q_0\} & \text{if } q = s, b = \varepsilon; \\ \{a\} & \text{if } q \in F, b = \varepsilon. \end{cases}$$

As the sole transitions in N that are not in D are null transitions, clearly $L(N) = L(D)$. Therefore, N and D are equivalent. □

Example 2.6.1. The next NFSA is the equivalent to automaton E in Example 2.2.1 that is described in Lemma 2.1. Let $\bar{E} = (\{a, b, s, f\}, \{0, 1, \varepsilon\}, \bar{\delta}, s, \{f\})$, where $\bar{\delta}$ is defined in Table 6.

Figure 15 is a digraph representation of this automaton.

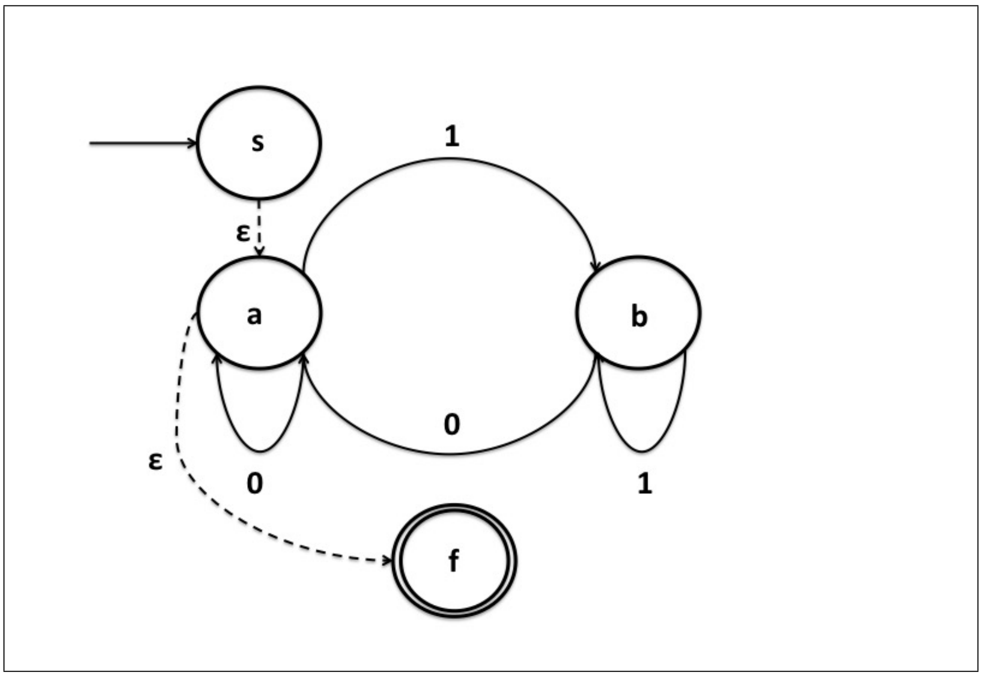

Figure 15: A nondeterministic automaton that is equivalent to automaton E

$\bar{\delta}$	0	1	ε
a	$\{a\}$	$\{b\}$	$\{f\}$
b	$\{a\}$	$\{b\}$	$\emptyset$
s	$\emptyset$	$\emptyset$	$\{a\}$
f	$\emptyset$	$\emptyset$	$\emptyset$

Table 6: Transitions of NFSA equivalent to E

2.7 Equivalence of NFSA and DFSA

In general, NFSA are not implementable, as there is no bound for the number of branches that a computation may have. Next we discuss a way around this problem.

Theorem 2.4. *Each NFSA has an equivalent DFSA.*

Proof. By construction. Given a nondeterministic NFSA $N = (Q, \Sigma, \delta, q_0, F)$, we construct a deterministic automaton $D = (Q', \Sigma, \delta', q'_0, F')$ such that $L(N) = L(D)$. This is done as follows:

1. Define $Q' = \mathscr{P}(Q)$
2. The transition map $\delta' : Q' \times \Sigma \to Q'$ is defined as

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a) \cup E(\delta(r, a)); \tag{2.2}$$

where, in general, for each $S \subset Q$

$$E(S) = \{q \in Q : \exists r \in S,\ q = \delta(r,\varepsilon)\}. \tag{2.3}$$

3. $q'_0 = \{q_0\}$, and

4. $F' = \{S \in Q' : S \cap F \neq \emptyset\}$.

Then, by construction, $L(N) = L(D)$. □

Remark 2.7.1. Equation 2.2 is motivated by the computation tree of N. In fact, each δ'-transition maps the set of states R into its descendants in the next level of the computation tree. Equation 2.3 adds all states that are reached from the latter states via null transitions.

In general, automaton D has exponentially more states than N, as $|Q'| = 2^{|Q|}$.

Example 2.7.1. Next is the deterministic equivalent of the nondeterministic automata introduced in Example 2.3.1.

1. $Q' = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$

2. $\delta' : Q' \times \{0,1\} \to Q'$ is defined in table 7

δ'	0	1	2
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{a\}$	$\{b,c\}$	$\emptyset$	$\emptyset$
$\{b\}$	$\emptyset$	$\{b,c\}$	$\emptyset$
$\{c\}$	$\emptyset$	$\emptyset$	$\{a\}$
$\{a,b\}$	$\{b,c\}$	$\{a,c\}$	$\emptyset$
$\{a,c\}$	$\{b,c\}$	$\emptyset$	$\{a\}$
$\{b,c\}$	$\emptyset$	$\{a,c\}$	$\{a\}$
$\{a,b,c\}$	$\{b,c\}$	$\{a,c\}$	$\{a\}$

Table 7: Definition of δ'

3. $q'_0 = \{a\}$, and

4. $F' = \{\{a\}, \{a,b\}, \{a,c\}, \{a,b,c\}\}$.

The computation of strings 0101 and 0200 are performed below:

$$\begin{aligned}
\{b,c\} &= \delta'(\{a\},0)\\
\{a,c\} &= \delta'(\{b,c\},1)\\
\{b,c\} &= \delta'(\{a,c\},0)\\
\{a,c\} &= \delta'(\{b,c\},1).
\end{aligned}$$

Since $\{a,c\} \in F'$, 0101 is accepted.

$$\begin{aligned}\{b,c\} =&\delta'(\{a\},0)\\ \{a\} =&\delta'(\{b,c\},2)\\ \{b,c\} =&\delta'(\{a\},0)\\ \emptyset =&\delta'(\{b,c\},0).\end{aligned}$$

Since $\emptyset \notin F'$, 0200 is rejected.

The next theorem is the reciprocal of Theorem 2.4

Theorem 2.5. *Each DFSA has an equivalent NFSA.*

Proof. By construction. Given $D = (Q,\Sigma,\delta,q_0,F)$, a deterministic FSA, we define the nondeterministic FSA $N = (Q,\Sigma,\delta',q_0,F')$

1. $\delta' : Q \times \Sigma \to \mathscr{P}(Q)$, where $\delta'(q,a) = \{r\}$ if and only if $\delta'(q,a) = r$; and

2. $F' = \{A \subset Q : A \cap F \neq \emptyset\}$.

Then, by construction, $L(N) = L(D)$. □

Corollary 2.2. *A language is regular if and only if it is recognized by a DFSA.*

2.8 Some Further Properties of Regular Languages

Theorem 2.6. *If L is a regular language over an alphabet Σ, then its complement over Σ^* is also a regular language.*

Proof. Since L is regular, there is a DFSA $D = (Q,\Sigma,\delta,q_0,F)$ that recognizes L. But then, the DFSA $M = (Q,\Sigma,\delta,q_0,Q-F)$ accepts a string if and only if D rejects it. Therefore, the complement $\bar{L} = \Sigma^* - L$ of L with respect to Σ, is recognized by M. Therefore, by Corollary 2.2, $\bar{L}$ is also regular. □

Corollary 2.3. *The intersection of two regular languages is a regular language.*

Proof. Let L and R be regular languages. Then, by Theorem 2.6, $\bar{L}$ and $\bar{R}$ are also regular languages. But then, $\bar{L} \cup \bar{R}$ is regular. By applying again Theorem 2.6 we conclude that

$$L \cap R = \overline{\bar{L} \cup \bar{R}};$$

is a regular language. □

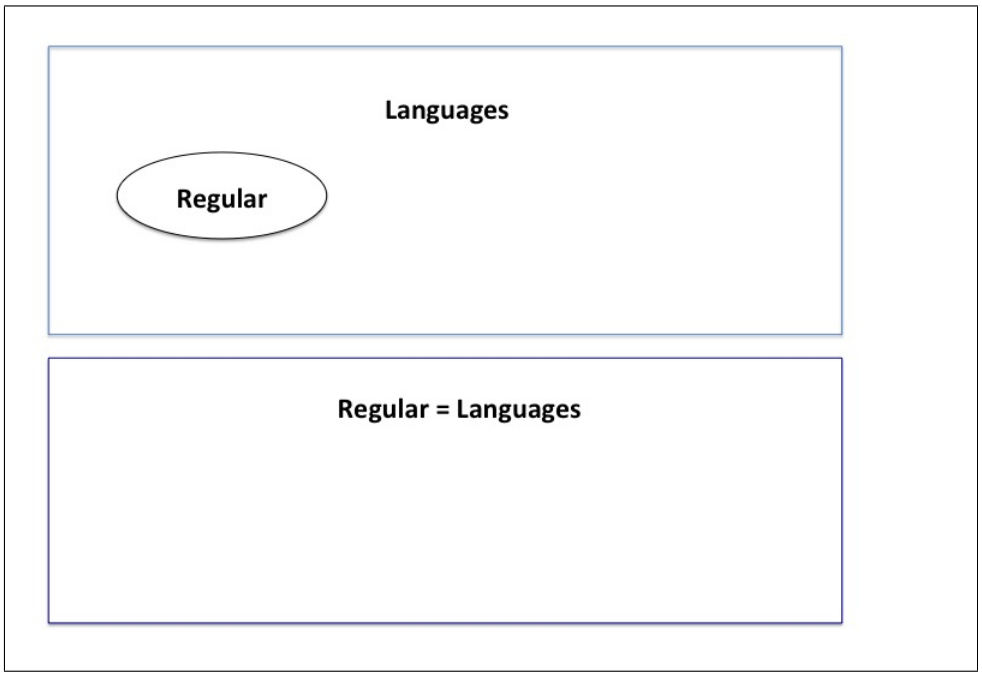

Figure 16: Venn diagrams of the possible answers to the question of the power of DFSA.

2.9 The Power of DFSA

A formal language L is a subset of the set of all strings over an alphabet Σ, this is, $L \subset \Sigma^*$. Or, what is the same, $L \in \mathscr{P}(\Sigma^*)$. Thus, we denote

$$Languages = \mathscr{P}(\Sigma^*).$$

Since DFSA solve only problems that can be encoded as regular languages, we asses the power of the DFSA model by denoting

$$Regular = \{L \in \mathscr{P}(\Sigma^*) : L \text{is regular}\},$$

and asking the question: Is $Regular = Languages$? Figure 16 presents this question as Venn diagrams.

Let's take a look into the next language.

Example 2.9.1. By an even palindrome we understand a string over an alphabet, which has even length and reads the same backwards or forward. For example, 01100110 is an even palindrome over $\Sigma = \{0,1\}$. Thus, we define

$$P2 = \{s \in \Sigma^* : s \text{ is an even palindrome}\}.$$

We notice that no regular operations are involved in the definition of a palindrome. In fact, any attempt to construct $P2$, or parse it in terms of regular operations, seems to fail. Is this just because it is hard to parse $P2$, or is it the case that $P2$ cannot be parsed?

The next lemma help answering this question.

Definition 2.9.1. We say that a language L satisfies the *Pumping Property I* if

$$(\exists p \in \mathbb{Z}, p > 0)(\forall s \in L)\,|s| \geq p \to (\exists x, y, z),\, s = xyz, \text{and}$$

1. $(\forall i \geq 0), xy^iz \in L$, and
2. $|y| > 0$, and
3. $|xy| \leq p$.

Lemma 2.2 (Pumping Lemma for Regular Languages)**.** *Every regular language satisfies the Pumping Property I.*

Proof. Let A be a regular language and $D = (Q, \Sigma, \delta, q_0, F)$ be a deterministic FSA with $L(D) = A$. Let $p = |Q|$ and $s = s_1 \cdots s_n$. Since $|s| \geq p$, the computation of s with D involves at least $p+1$ states, and therefore, at least one state r is repeated in this computation. In particular, this means that there are positive integers m and q, with $m < q < p$, such that

$$r = \delta(s_m, q)$$

is the first appearance of r in the computation of s; and

$$r = \delta(s_q, t)$$

is the second. Then take,

$$\begin{aligned} x &= s_1 \cdots s_m \\ y &= s_{m+1} \cdots s_q, \text{ and} \\ z &= s_{q+1} \cdots s_n. \end{aligned}$$

Clearly, this segmentation of s satisfies conditions 2 and 3 of the Lemma. And since the computation from s_m to s_q is a loop, any string of the form xy^iz will be accepted by D, including the case $i = 0$, which corresponds to xz. Thus, for all i, $xy^iz \in L(D) = A$. □

Remark 2.9.1. It is important to notice that Lemma 2.2 states that all regular languages satisfies the Pumping Property I but it does not state that this property is exclusive of regular languages. Indeed, as we will see in Section 2.10 there are non regular languages that also satisfy it. Thus, the Pumping Lemma for Regular Languages cannot by used to prove that a language is regular. But, quite the opposite, it is the perfect instrument to prove that a language is not regular, since its contrapositive states that if a language fails to satisfy the Pumping Property I, then is not a regular language. Consequently, the negation of the Pumping Property I is a test of the regularity of a language.

Let's examine the negation of the Pumping Property I with basic predicate logic,

$$\begin{aligned}
&\neg((\exists p \in \mathbb{Z}, p > 0)(\forall s \in A)(|s| \geq p) \to (\exists x, y, z), s = xyz \wedge (1) \wedge (2) \wedge (3))\\
\leftrightarrow &(\forall p \in \mathbb{Z}, p > 0)(\exists s \in A)(|s| \geq p) \wedge \neg((\exists x, y, z), s = xyz \wedge (1) \wedge (2) \wedge (3))\\
\leftrightarrow &(\forall p \in \mathbb{Z}, p > 0)(\exists s \in A)(|s| \geq p) \wedge ((\forall x, y, z)\, s \neq xyz \vee \neg(1) \vee \neg(2) \vee \neg(3).
\end{aligned}$$

Here (1), (2) and (3) refer to properties 1, 2 and 3 in Definition 2.9.1, respectively. For using this negation as a test we assume first that

$$(\forall x, y, z)\, s = xyz \wedge |y| > 0 \wedge |xy| \leq p; \tag{2.4}$$

which renders all statements in the negation of Pumping Property I false, except by $\neg(1)$. Consequently, if 2.4 is assumed, we need to prove that $\neg(1)$ is true, or the negation will not hold. This amounts to showing that

$$(\exists i > 0) xy^i z \notin L.$$

In summary, our method for proving that a language L is not regular consists in finding a string s that depends on a variable p ranging over the positive naturals, and prove that s satisfies (1) and (2) but for each segmentation $s = xyz$ there always exist i such that $xy^i z \notin L$. When this is the case, we say that s cannot be pumped.

Theorem 2.7. *The language of all even palindromes over an alphabet* Σ

$$P2 = \{s \in \Sigma^* : (\exists q \in \mathbb{N})\, s = s_1 s_2 \cdots s_q s_q s_{q-1} \cdots s_1\},$$

is not regular.

Proof. For each p, $p \in \mathbb{N}$, take a string of the form $s = s_1 \cdots s_p s_p \cdots s_1$. Clearly $|s| > p$ and for each segmentation $s = xyz$ that satisfies $|y| > 0$ and $|xy| \leq p$, y is a substring of $s_1 \cdots s_p$. So,

$$s = xyz = s_1 \cdots y \cdots s_p s_p \cdots \bar{y} \cdots s_1;$$

where $\bar{y}$ is y in reverse order. Now,

$$xy^2 z = s_1 \cdots yy \cdots s_p s_p \cdots \bar{y} \cdots s_1;$$

and thus, $xy^2 z$ is not a palindrome. □

Therefore, *Regular* $\subset$ *Languages* and as Figure 17 suggests, the set *Languages* is in fact much bigger than *Regular*.

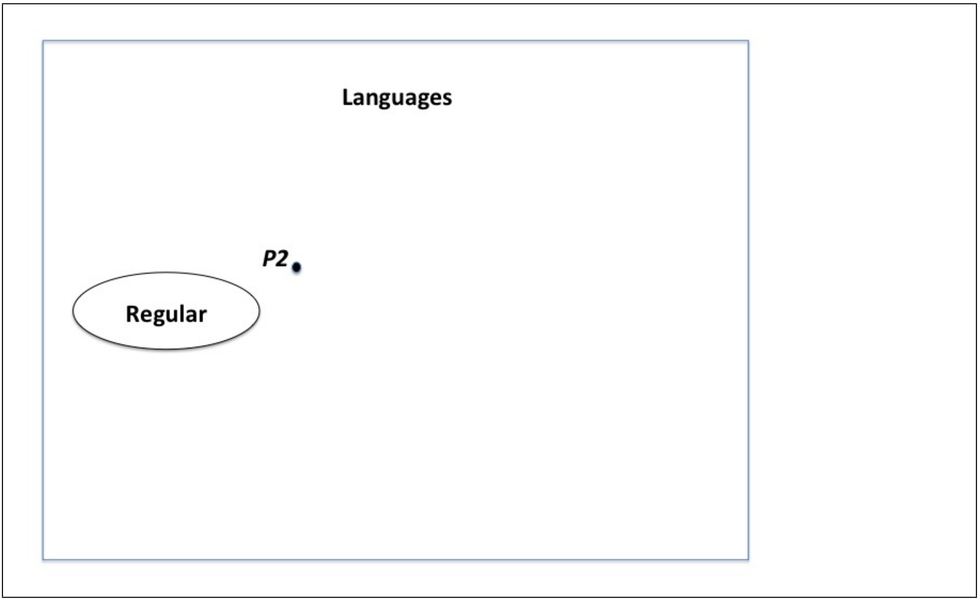

Figure 17: $P2$ is not a regular language.

2.10 A Non-regular Language Satisfying PPI

Next we show a non-regular language that satisfies the Pumping Property I.

Theorem 2.8. *Let*

$$L = \{w \in \{0,1,2\}^* : w = 0^i 1^j 2^k \text{ where } (\forall i, j, k \in \mathbb{N}), i = 1 \to j = k\}.$$

Then, L satisfies PPI.

Proof. Let $p = 2$. For each $s \in L$ with $|s| \geq 2$, we consider two cases.

1. If $i \neq 2 \wedge i \neq 0$, we select $x = \varepsilon$, $y = 0$ and z the rest of s,
2. If $i = 2$, we take $x = \varepsilon$, $y = 00$ and z the rest of s,
3. If $i = 0$, we select $x = \varepsilon$, $y = 1$ and z the rest of s, and
4. If $i = j = 0$, we select $x = \varepsilon$, $y = 2$ and z the rest of s.

In all cases, $|y| > 0$ and $|xy| \leq p$. Now, if $i \neq 2 \wedge i \neq 0$, for each $l \geq 0$,

$$xy^l z = 0^{l+i} 1^j 2^k.$$

Since $i + l > 1$, $xy^l z \in L$. On the other hand, if $i = 2$, since $y = 00$, for each $l \geq 0$,

$$xy^l z = 0^{2l} 1^j 2^k,$$

which is also a string in L.

In the cases $i = 0$ and $i = j = 0$, the rule

$$(\forall i, j, k \in \mathbb{N}), i = 1 \to j = k$$

does not apply, and therefore, all pumped strings are members of L, as well.

Thus, L satisfies PPI. □

Remark 2.10.1. Although not explicit in the proof of Theorem 2.8, a different segmentation is necessary for the cases $i \neq 2 \wedge i \neq 0$, and $i = 2$. Indeed, if $y = 0$ when $s = 001^j2^k$, in the case in which $l = 0$ in xy^lz produces the string $0^001^j2^k$. Now, since $0^0 = \varepsilon$,

$$0^l01^j2^k = 0^001^j2^k = 01^j2^k,$$

which is not a member of L unless $j = k$.

Next we demonstrate that L is not regular. The demonstration follows as a corollary of the following lemma.

Lemma 2.3. *The language*

$$R = \{01^j2^k : \ j, k \in \mathbb{N}\}$$

is regular, but the language

$$N = \{01^n2^n : \ n \in \mathbb{N}\},$$

is not regular.

Proof. Clearly, language R is represented by the regular expression 01^*2^*. So, R is regular. To show that N is not regular, for each $p \in \mathbb{N}$, $p > 0$, set $s = 01^p2^p \in N$. For each segmentation $s = xyz$, with $|y| > 0$ and $|xy| \leq p$, xy is a substring of 01^n. Consequently, any pumping of y produces a string that is either not of the form 01^n2^n, or one that has more $1'$s than $0'$s. Thus, $xy^lz \notin N$. □

Corollary 2.4. $L = \{w \in \{0,1,2\}^* : w = 0^i1^j2^k \text{ where } (\forall i, j, k \in \mathbb{N}), i = 1 \to j = k\}$ *is not a regular language.*

Proof. By contradiction. Assume that L is regular. Then, using the previously defined languages R and N, we have

$$L \cap R = \{01^n2^n : \ n \in \mathbb{N}\} = N.$$

Since R is regular, N is the intersection of two regular languages. Using Corollary 2.3 we conclude that N is also a regular language. But it was proved in Lemma 2.3 that N is not regular. Therefore, the assumption that L is regular is false. □

Chapter 3

Context-free Languages

This chapter concentrates in the description and automated recognition of a string in a *Context-free Language* (CFL). The class of all CFL includes the regular languages and some non-regular ones, such as the set of all even palindromes. So, in that sense, the CFL construction and string recognition mechanisms are more powerful than those used to construct and recognize strings in regular languages. But there is a price to pay for the extra power. As it will be apparent in this chapter, the theory of CFL generation and string recognition is not as complete nor as clean and simple as that of regular languages.

3.1 Context-free Grammars

The basic operations for constructing strings in a CFL are defined next.

Definition 3.1.1. A Context-free Grammar (CFG) over an alphabet Σ is a 4-tuple $G = (V, \Sigma, R, S)$ where

1. Σ is the alphabet of the grammar. In the context of grammars, the elements of Σ are called *terminals*;
2. V is a finite set of symbols called *variables*, which are selected in such a way that $V \cap \Sigma = \emptyset$;
3. R is a finite set of *rules*; and
4. $S \in V$ is the start variable.

The set of rules is a relation $R \subset V \times (\Sigma \cup V)^*$, whose pairs (v, ω) are denoted $v \rightarrow \omega$.

Example 3.1.1. We adopt the convention of denoting an instance of two or more rules starting with the same variable V, let's say, $V \rightarrow \omega$, $V \rightarrow \gamma$ as $V \rightarrow \omega|\gamma$. Then, the 4-tuple

$$G = (\{S, U\}, \{0, 1\}, \{S \rightarrow 0U0|1U1, U \rightarrow 0U0||1U1|\varepsilon\}, S),$$

is a CFG over $\Sigma = \{0,1\}$.

Definition 3.1.2. Given two rules $v \to \omega_1 \cdots \omega_{k-1} z \omega_{k+1} \cdots \omega_n$ with z a variable, and $z \to \gamma_1 \cdots \gamma_m$ in a CFG $G = (V, \Sigma, R, S)$, the operation

$$v \to \omega_1 \cdots \omega_{k-1} \underline{z} \omega_{k+1} \cdots \omega_n \to \omega_1 \cdots \omega_{k-1} \underline{\gamma_1 \cdots \gamma_m} \omega_{k+1} \cdots \omega_n,$$

is called a *substitution*.

A *derivation*, or more explicitly, a *G-derivation*, is a finite sequence of substitutions that start with a rule of the form $S \to \omega$ and ends with a string in Σ^*.

Example 3.1.2. Here are two derivations with the CFG defined in Example 3.1.1:

1. $S \to 0U0 \to 0\varepsilon0$ produces 00.

2. $S \to U \to 0U0 \to 01U10 \to 01\varepsilon10$, produces 0110.

3.2 The Language of a CFG

Definition 3.2.1. The language of a CFG $G = (V, \Sigma, R, S)$, denoted $L(G)$, is defined as

$$L(G) = \{s \in \Sigma^* : \ s \text{ is produced with a } G-derivation\}.$$

A language L for which there is a CFG G such that $L = L(G)$ is called *context-free language* (CFL).

Proving the regularity of a language is a well-defined process that can be implemented as an algorithm. This is not the case of Context-free Languages. In general, proving that L is a CFL requires a mathematical prove of the predicate

$$(\exists G, \ G \text{ a context} - \text{free grammar }) \ L = L(G).$$

Depending on L, this prove may range from rather simple to very hard. Mathematical induction is a recurrent method of proof in these endeavors. As a way of illustration, we demonstrate that the language of the CFG defined in Example 3.1.1 is $P2$. For this purpose, we introduce the next auxiliary concept.

Definition 3.2.2. Let n be an odd natural number. A string $w = w_1 \ldots w_n$ is said to be *odd-symmetric* if and only if

$$(\forall i, j \in \{1, ..., n\}) \ i + j = n + 1 \to w_i = w_j.$$

Lemma 3.1. *Let G be the CFG in Example 3.1.1. Then, each sequence of substitutions generates either an odd symmetric string* $w = w_1 ... w_n \in (\{0,1\} \cup \{S,U\})^*$ *of literals, except for* $w_{\frac{n+1}{2}}$ *which is a variable, or an element in* $P2$.

Proof. By induction on the number n of substitutions.
Base case. If $n = 1$, the only rule that is available is $S \to U$. Now, U is an odd-symmetric string over $(\{0,1\} \cup \{S,U\})^*$ with a single variable in position $1 = \frac{1+1}{2}$.
Induction. Assume that the statement is true for $n = k$ steps. Thus, the string generated with k substitutions is either of the form $w = z\bar{z} \in P2$ or $w = zZ\bar{z}$ with $Z \in V$. In the first case, $w \in P2$. In the second, because of the definition of R, $Z = U$ necessarily. Thus, the rules that are available are either $U \to \varepsilon$ or $U \to 0U0|1U1$. In the first case, the $k+1$ substitution produces $z\bar{z} \in P2$ and in the second, is either $z0U0\bar{z}$ or $z1U1\bar{z}$, which is an odd-symmetric string of the type specifed in the statement of the Lemma. □

Corollary 3.1. *Let G be the CFG in Example 3.1.1. Then, $L(G) = P2$.*

3.3 Relation with Regular Languages

We demonstrate now that the class of all regular languages is a subset of the class of all context-free languages.

Theorem 3.1. *Each regular language is a context-free language.*

Proof. Let L be a regular language over an alphabet Σ, and let R be its regular expression. Assume for simplicity that $R = R_1 \cup R_2 \cup \ldots \cup R_n$ is the expression of R in conjunctive normal form. Then, the next procedure produces a context-free grammar for generating L.

1. For each $i = 1,\ldots,n$, set the rules

$$S \to V_1|\ldots|V_n.$$

   where S is the start variable and V_i, $i = 1,\ldots,n$ variables, and for each $i = 1,\ldots,n$ the rule

$$V_i \to a_i Z$$

   where $a_i \in \Sigma$ is the leftmost element in R_i.

2. Read R_i from left to right and for each instance of $b_1 \cdots b_k \in \Sigma^*$ produce the sequence of rules

$$Z_j \to b_j Z_{j+1},\ j = 1,\ldots,k+1.$$

   If b_k is the last symbol of Σ in R_i, add the rule $Z_k \to \varepsilon$. Else, add the rule $Z_{k+1} \to cU$ where $c \in \Sigma$ is the first element to the right of $b_1 \cdots b_k$ in R_i.

   For each $(b_1 \cdots b_k)^*$ produce the previous sequence of rules and add the next rules

$$Z_{k+1} \to Z_1 \text{ and } Z_1 \to \varepsilon.$$

3. If b_k is the last element of Σ in R_i, add the rule

$$Z_{k+1} \to \varepsilon.$$

Clearly, the context-free grammar constructed with these rules generates the language represented by R. □

Example 3.3.1. We construct a CFG for generating the regular language represented by $R = a(bc)^* \cup cb$.
R is in conjunctive normal form with $R_1 = a(bc)^*$ and $R_2 = cb$. Thus, we set first $S \to V_1 | V_2$ and

$$V_1 \to aZ_2$$
$$V_2 \to cW_2.$$

Since V_1 and V_2 are defined with the leftmost elements in R_1 and R_2, respectively, we denote V_1 as Z_1 and V_2 as W_2 for notational consistency. Now, by reading R_1 we get

$$Z_1 \to aZ_2$$
$$Z_2 \to bZ_3$$
$$Z_3 \to cZ_4.$$

Since string segment bc is under a Kleene star, we add the rules

$$Z_4 \to Z_2,$$
$$Z_2 \to \varepsilon.$$

And since c is the last element in R_1, we add

$$Z_4 \to \varepsilon.$$

By reading R_2 we get

$$W_1 \to cW_2,$$
$$W_2 \to bW_3.$$

And since b is the last element in R_2, we add

$$W_3 \to \varepsilon.$$

3.4 Chomsky's Normal Form

Definition 3.4.1. Two CFGs G_1 and G_2 are said to be *equivalent* if and only if $L(G_1) = L(G_2)$.

Definition 3.4.2. A CFG is in *Chomsky's Normal Form* (CNF) if and only if its rules are all of the form of either

$$A \to BC,$$

where A, B and C are variables, but neither B nor C are the start variable,

$$A \to \alpha,$$

where $\alpha \in \Sigma$ is a terminal symbol; or

$$S \to \varepsilon$$

where S is the start variable.

Theorem 3.2. *Each CFG has an equivalent CFG in CNF.*

Proof. Given a CFG $G = (V, \Sigma, R, S)$, the next procedure creates a CFG in CNF that is equivalent to G.

1. Create a new start variable S_0 and the rule $S_0 \to S$.
2. Remove each rule of the form $A \to \varepsilon$, where A is not the start variable. In order to preserve $L(G)$, for each ocurrence of A in rule of the form of $B \to uAv$, for each each rule $A \to w$ add a new rule $B \to uwv$.
3. Eliminate all rules of the form $A \to B$. In order to get preserve $L(G)$ whenever a rule $B \to u$ appears, add a rule $A \to u$ unless the new rule equals one that was already eliminated in Step 2.
4. Convert all remaining rules in rules of the form $A \to BC$ or $A \to a$. In order to get an equivalent grammar, each rule of the form $A \to u_1 \cdots u_k$ is expressed as

$$\begin{aligned} A &\to u_1 A_1, \\ A_1 &\to u_2 A_2 \\ &\vdots \\ A_{k-2} &\to u_{k-1} u_k; \end{aligned}$$

   where A_i, $i = 1, ..., k-2$ is a new variable. Then, replace any terminal u_i in the preceding rules with a new variable U_i and add the rule $U_i \to u_i$.

Clearly, the resulting CFG is in CNF. Since each step in this procedure preserves $L(G)$, the new grammar is equivalent to G □

Example 3.4.1. Next we find a context-free grammar in Chomsky normal form, that is equivalent to the context-free grammar introduced in Example 3.1.1.

1. Introduction of a new start variable and rule.

$$\begin{aligned} S_0 &\to S \\ S &\to 0U0|1U1 \\ U &\to 0U0|1U1|\varepsilon \end{aligned}$$

2. Remotion of $U \rightarrow \varepsilon$.

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow 0U0|1U1|00|11 \\ U &\rightarrow 0U0|1U1|00|11 \end{aligned}$$

3. Elimination of the rule $S_0 \rightarrow S$.

$$\begin{aligned} S_0 &\rightarrow 0U0|1U1|00|11 \\ U &\rightarrow 0U0|1U1|00|11 \end{aligned}$$

4. We modify first the rules of the form of $A \rightarrow ab$ with a and b literals.

$$\begin{aligned} S_0 &\rightarrow 0U0|1U1|U_1U_1|U_2U_2 \\ U &\rightarrow 0U0|1U1|U_1U_1|U_2U_2 \\ U_1 &\rightarrow 0 \\ U_2 &\rightarrow 1. \end{aligned}$$

   Next we modify the rules of the form $A \rightarrow aBc$ with a and b literals, and B variable.

$$\begin{aligned} S_0 &\rightarrow U_1Z_1|U_2Z_2|U_1U_1|U_2U_2 \\ U &\rightarrow U_1Z_1|U_2Z_2|U_1U_1|U_2U_2 \\ U_1 &\rightarrow 0 \\ U_2 &\rightarrow 1 \\ Z_1 &\rightarrow UU_1 \\ Z_2 &\rightarrow UU_2, \end{aligned}$$

   which is in CNF and is equivalent to the CFG in Example 3.1.1.

Let's now use this grammar to generate the string $010010 \in P2$.

$$\begin{aligned} S_0 &\rightarrow U_1Z_1 \\ &\rightarrow U_1UU_1 \\ &\rightarrow U_1U_2Z_2U_1 \\ &\rightarrow U_1U_2UU_2U_1 \\ &\rightarrow U_1U_2U_1U_1U_2U_1 \rightarrow 010010. \end{aligned}$$

In this derivation we have applied first all the substitutions that used rules of the form $A \rightarrow BC$ and then, all substitutions that use rules of the form $A \rightarrow a$. Although the same string can be generated applying the substitutions in a different order, this particular order illustrates the proof of the next Lemma.

Lemma 3.2. *A context-free grammar in Chomsky normal form generates a string of length n with exactly* $2n-1$ *substitutions.*

Proof. If string $w = w_1...w_n$ is generated by a context-free grammar G in Chomsky normal form, then for each $i = 1,...,n$ G has a rule $V_i \rightarrow w_i$, where some V_i's may be repeated. So, generating w form $V_1 \cdots V_n$ takes n substitutions. Now, since G is in Chomsky normal form, $V_1 \cdots V_n$ is generated with substitutions of the form of $U_i \rightarrow V_iV_{i+1}$, $i = 1,...,n-1$. Thus, generating $V_1 \cdots V_n$ requires exactly $n-1$ substitutions, which proves the statement. □

3.5 Pushdown Automata

A *Pushdown Automata* (PDA) is an abstract model of an automatic process to decide whether a string was generated by a grammar. Making such decision requires the parsing of the string to determine if its structure is consistent with that of a string generated by the grammar. It turns out that string parsing is the main difference between FSAs and PDAs. As we saw in the previous chapter, the regular expression of a language can be *hardwired* into an FSA and thus, everything the FSA needs to do to decide if a string is in the language is matching it to the hardwired structure. CFLs do not have a description as powerful as a regular expression. Deciding wether a string is or is not a member of a CFL involves a non-deterministic parsing and search for a form that is consistent with a derivation of the grammar. This search may eventually require some data storage. PDAs use a *stack memory* model, which avoids the overheads of addressing.

Definition 3.5.1. A *pushdown automaton* (PDA) is a six-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

1. Q is a finite set of state labels,
2. Σ is the alphabet of the language,
3. Γ is the *stack alphabet*,
4. $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\} \rightarrow \mathscr{P}(Q \times \Gamma \cup \{\varepsilon\})$ is the transition relation,
5. $q_0 \in Q$ is the start state, and
6. $F \subset Q$ is the set of accept or final states.

Thus, each transition takes a triple $(q, a, \alpha) \in Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\}$ into a set of pairs

$$\delta(q, a, \alpha) = \{(r, \beta) : r \in Q, \beta \in \Gamma \cup \{\varepsilon\}\} \tag{3.1}$$

The stack is a string $\gamma \in \Gamma^*$. The transition (3.1) exhibits only the leftmost elements in γ, in this case, α before the reading of $a \in \Sigma \cup \{\varepsilon\}$, and β, after the reading. Thus, the reading induces a two step operation, namely, the deletion of α

(a *pop* out of the stack), followed with the concatenation of β to the left end of γ (a *push* into the stack). Notice that this pop-and-push operation cannot be executed if when reading a in state q, α is not the leftmost character in the stock string γ.

Definition 3.5.2. A computation of a PDA is a rooted tree of pairs $(q,\alpha) \in Q \times \Gamma^*$, with root (q_0,ε), that depicts the state transitions and changes in the stack generated by the reading of an input string. Each edge

A string is accepted if its computation tree has (q,ε), $q \in F$, as a leaf.

Remark 3.5.1. The nodes in a PDA computation tree can be reduced to state and top-of-stack pairs. However, as shown below, state and top-of-stack pairs are not enough to defined the transitions in a lookup table, as some pop operands may be left unespecified. This deficiency is remedied by labelling the edges of a transition graph with the character read and the pop and push operands. In general, it is preferable to use a *high level description.* This is a description in natural language of the actions of the PDA, pretty much in the style of a pseudo code.

Example 3.5.1. Next is the high-level description of a PDA for recognizing $P2$, the language of even palindromes over $\Sigma = \{0,1\}$.

The idea behind this pushdown automaton is to decide whether the input string is or not odd-symmetric, by pushing it first half into the stack, and then popping it out of the stack. Problem is that the point in the execution where the first half of the string is being read is unknown. The method guesses a mid point and start checking the odd-symmetry. In a high level descriptions the term *nondeterministically* is used to refer to nondeterministic choices in the execution of the automaton.

Algorithm 1 Pushdown automaton for recognizing $P2$

```
1: procedure (Input: $w$)
2:     Push \$ into the stack.
3:     Start reading $w$ from left to right pushing each read symbol into the stack.
4:     Nondeterministically, replace the push operation with the pop operation.
5:     If $w$ is read and \$ is the top of a stack, empty the stack and accept $w$.
```

If none of the nondeterministic replacements leads to the acceptance of the input string, the string is rejected.

It is worth remarking that, if the input string is of even length, among all possible nondeterministic selections of change from push to pop, there is only one that occurs right after the first half of the string has been read. If the string is odd-symmetric, this choice provokes ending the reading with the symbol \$ in the top of its stack. Only in that case, the automaton empties the stack and accepts. However, if this is not the case, the string cannot be rejected. As pointed out above, to reject it, all possible *guesses* must have failed.

Next is an alternative description based on a graph of transitions. Let

$$P = (\{q_0,q_1,q_2,q_3\},\{0,1\},\{\$,0,1\},\delta,q_0,\{q_3\}),$$

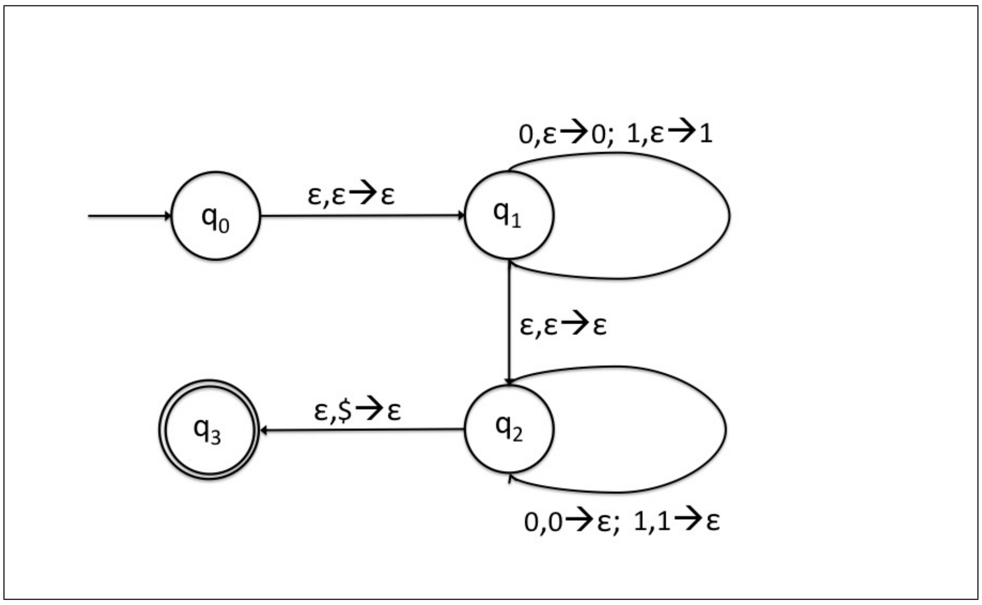

Figure 18: Diagram of transitions for pushdown automaton P

whose transitions δ are defined in Figure 18. Here, the actions of the pop of a out the stack followed by the push of b; $a,b \in \Sigma \cup \{\varepsilon\}$, is represented on the edge of the corresponding transition as $a \to b$.

Figure 19 is the computation tree of string $w = 0110$.

3.6 The Language of a Pushdown Automaton

Definition 3.6.1. The language of a pushdown automaton A is the set $L(A)$ of all the strings that the automaton accepts. As before, a language L is said to be recognized by A, if $L = L(A)$.

Theorem 3.3. *If L is a Context-free language, then there is a pushdown automaton A that recognizes it.*

Proof. Let L be a context-free language. Then, there is a CFG (V,Σ,R,S) that generates L. The proof is again by construction of a PDA that recognizes L. This automaton is described in Algorithm 2.

The PDA described in Algorithm 2 nondeterministically searches for a G-derivation of the input string. Indeed, each nondeterministic decision made by the PDA described in Algorithm 2 guesses a substitution in the sequence of substitutions that constitute that derivation. Again, if the string is in the language, the PDA will guess the right sequence of substitutions, and accept the string. Instead, if all combinations of nondeterministic choices of substitution fail to create a derivation of the input string, the string is not generated by G, and must be rejected. □

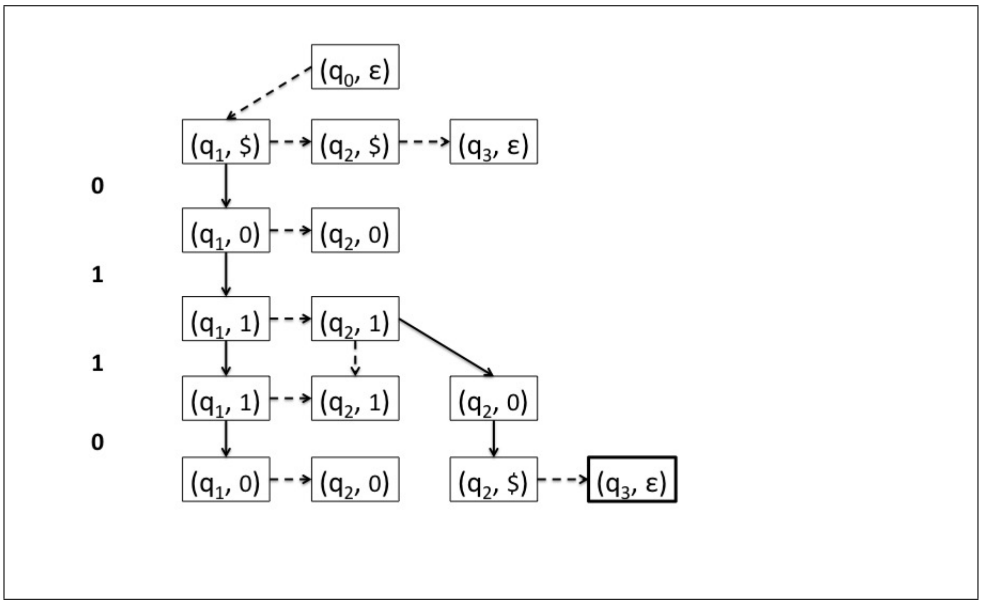

Figure 19: Computation tree of string $w = 0110$ with pushdown automaton P. Segmented arrows represent ε transitions.

Lemma 3.3. *Each pushdown automaton is equivalent to a pushdown automaton that*

1. *the first step of each computation, just pushes* \$ *into the stack;*
2. *has a single accept state; and*
3. *at each transition, either pushes a symbol into the stack or pops a symbol out, but never do both.*

Proof. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a pushdown automaton. To prove Claim 1, we define a new start state q_{start}, and the transition

$$\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_0, \$)\}.$$

To prove Claim 2, we define a new accept state symbol, q_{accept} and the transitions

$$\delta(q, \varepsilon, \varepsilon) = \{(q_{accept}, \varepsilon)\},\ \forall q \in F.$$

and then define $F = \{q_{accept}\}$.
To prove Claim 3, we accompany each transition $(r, \beta) \in \delta(q, a, \alpha)$ with the graph description of the pop and push sequence, $a, \alpha \to \beta$. Now, if $\alpha = \varepsilon$ or $\beta = \varepsilon$, but not both, the transition has already the required property. If $\alpha = \varepsilon$ and $\beta = \varepsilon$, we introduce a new symbol $\gamma \in \Gamma$, a new state $s \in Q$; and define the transitions,

$$\delta(s, \gamma) \in \delta(q, a, \varepsilon)\ \wedge\ \delta(r, \varepsilon) \in \delta(s, \varepsilon, \gamma),\ \text{so that}$$
$$a, \varepsilon \to \gamma,\ \wedge\ \varepsilon, \gamma \to \varepsilon.$$

Algorithm 2 Pushdown automaton for recognizing a string generated by G

```
1:  procedure (Input: $w = w_1 \cdots w_n, G = (V, \Sigma, R, S)$)
2:      Push \$ and $S$ into the stack.
3:      Read $w$ from left to right.
4:      for each symbol $a$ in $w$ do
5:          Read the top of the stack.
6:          if the top of stack is a variable $U \in V$ then
7:              Nondeterministically select a rule $U \to \alpha$.
8:              Pop $U$.
9:              Read $\alpha$ from right to left pushing each element into the stack.
10:         if the top of stack is a literal $b \in \Sigma$ then
11:             if $b = a$ then
12:                 Pop $b$.
13:                 Read next symbol in $w$.
14:         if the top of stack is \$ then
15:             Empty the stack and accept.
```

Finally, if both, $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, define a new state s and the transitions

$$\delta(s, \varepsilon) \in \delta(q, a, \alpha) \wedge (r, \beta) \in \delta(s, \varepsilon, \varepsilon), \text{ so that}$$
$$a, \alpha \to \varepsilon, \wedge \varepsilon, \varepsilon \to \beta;$$

Clearly, neither of the introduced modification alters the computation with P. □

Theorem 3.4. *If A is a pushdown automaton, then $L(A)$ is a context-free language.*

Proof. Let B be a PDA equivalent to A constructed with the specifications given in the proof of Lemma 3.3. Thus, in particular, $L(B) = L(A)$. Now, the computation of each $w = w_1 \cdots w_n \in L(B)$ produces at least one accepting computation path. We represent this path as an ordered tuple of triplets (q, a, h); where q is a state in that path, a is the input symbol at state q, or ε; and h is the length of the stack string at state q. By numbering the state in the path with the levels of depth of the computation tree, we get a tuple representation of the form of

$$S = ((q_{st}, \varepsilon, 0), (q_1, w_1, 1), ..., (q_i, w_i, h), ..., (q_n, w_n, 1), (q_{ac}, \varepsilon, 0)),$$

for the computation of w. Let L be the maximum length that the stack of B reaches in the selected accepted computation of w. Then, for each $l = 0, ..., L$ we define

$$H(l) =< i : \exists (q_i, w_i, l) \in A >=< i_k^l : k = 1, ..., m >$$

where the brackets $< >$ denote an ordered list, sorted in the increasing order of the indices i. With these elements, we define the Algorithm 3. This method takes an string accepted by B and returns a list of rules for deriving it.

The proof of the existence of a context-free grammar for generating $L(B)$ is a non-constructive proof of existence. This is, we show that such context-free grammar exists, without constructing it. Indeed, for each $w \in L(B)$, Algorithm 3 finds a set of rules for generating w. Each rule is either of the form of

$$U \to \varepsilon; \tag{3.2}$$
$$U \to aZb,\ a,b \text{ literals; or} \tag{3.3}$$
$$U \to ZW. \tag{3.4}$$

Since each variable is in one-to-one correspondence with a pair of states, there are at most $|Q|^2$ variables. Since each rule in (3.2) is in one-to-one correspondance with a variable, there are at most $|Q|^2$ rules of this type. A similar argument shows that there are at most $|Q|^4|\Sigma|^2$ rules of type (3.3); and since each rule of the form of (3.4) involves three variables and no literals, there are at most $(|Q|^2)^3 = |Q|^6$ rules of the type of (3.4). Therefore, the number of non-repeated rules is bounded above by $|Q|^2 + |\Sigma|^2|Q|^4 + |Q|^6$. This proves the existence of a finite set of rules R for producing the strings in $L(B) = L(A)$. Therefore, there is a context-free grammar $G = (V, \Sigma, R, V_{s,a})$, $V \subset \{V_{qr} : q, r \in Q\}$, that produces the language $L(A)$. Hence, $L(A)$ is a context-free language.

Algorithm 3 Computation of rules for generating w

```
1: procedure ( Input: $S$ )
2:     Compute $L = \max\{h : (q,a,h) \in S\}$.
3:     $H \leftarrow \emptyset$
4:     for $l = 0,...,L$ do
5:         Compute $H(l) =< i : (q_i, w_i, l) \in S >$
6:         $H \leftarrow H \cup H(l)$.
7:     for l = 0,...,L do
8:         if $H(l) =< i_1^l,...,i_k^l >$ has more than two elements then
9:             Split into $H_1(l) =< i_1^l, i_2^l >$, .... , $H_{r-1}(l) =< i_{r-1}^l, i_r^l >$.
10:            Set the rule $V_{i_1^l i_r^l} \to V_{i_1^l i_2^l} V_{i_2^l i_3^l} \cdots V_{i_{r-1}^l i_r^l}$
11:    for l = 0,..., L do
12:        for each $H_k(l)$ do
13:            $n \leftarrow i_k$, $m \leftarrow i_{k+1}$
14:            while $m - n > 2$ do
15:                Set the rule $V_{n,m} \to w_n V_{n+1,m-1} w_{m-1}$
16:                $n \leftarrow n+1$, $m \leftarrow m-1$
17:            Set the rule $V_{n,n} \to \varepsilon$
```

□

Corollary 3.2. *Each context-free language can be generated with a context-free* *grammar whose rules are either*

$$\begin{aligned}&V \to \varepsilon,\\ &V \to aUb,\ \text{a, b literals, and}\\ &V \to UZ.\end{aligned}$$

Proof. Is a direct consequence of the proof of Theorem 3.4. □

3.7 Existence of Non Context-free Languages

Definition 3.7.1 (Pumping Property II)**.** A language L satisfies the Pumping Property II (PPII) if and only if $(\exists p \in \mathbb{N}), p > 0$, $(\forall s \in L)$

$$|s| \geq p \to (\exists u, v, x, y, z \in \Sigma^*)\ s = uvxyz$$

that satisfies

1. $(\forall i \geq 0, i \in \mathbb{N})\ uv^ixy^iz \in L$;
2. $|vy| > 0$; and
3. $|vxy| \leq p$.

Theorem 3.5 (Pumping Lemma for Context-free Languages)**.** *If L is a Context-free language, then L satisfies the Pumping Property II.*

Proof. Since L is a context-free language, there is a CFG $G = (V, \Sigma, R, S)$ that generates L. We assume that G is in CNF. Therefore, the parse tree of a string derived by G, is a binary tree. In particular, if $|s| \geq p$ and λ is the length of the longest path in the derivation of s, then $\lambda \geq \log_2(p)$. Since the last node in the longest path is always a literal, let's define $p = 2^{|V|+2}$. Then, $\lambda \geq |V| + 2$ and the path has at least $|V| + 1$ variables. Thus, there is a variable A that is repeated in the path. This allow a derivation of s of the form of

$$S \to \cdots \to uAv \to \cdots \to uvAyz \to \cdots \to uvxyz, \tag{3.5}$$

where $s = uvxyz$. Thus, there is a derivation segment of the form $A \to \cdots vAy$, which may be applied recursively, to get

$$S \to \cdots \to uAv \to \cdots \to uvAyz \to \cdots \to uv^2Ay^2z \to \cdots \to uv^ixy^iz,$$

after i recursive applications, $i \in \mathbb{N}$. Consequently, each string of the form uv^ixy^iz is generated by G and thus, is a member of L. This shows that L satisfies item 1 in PPII. Regarding item 2, it is clear that is not possible for v and u to be the null string because if this were the case, the rule $A \to A \in R$ but this kind of rules are not allowed in a CFG in CNF. As for item 3, we assume that the repetitions in

Equation (3.5) are selected in such a way that there are no other repeated variables in the derivation segment

$$uAv \to \cdots \to uvAyz \to \cdots \to uvxyz.$$

Thus, the length of the corresponding path segment is less that or equal to $|V|$, and therefore it cannot generate a string longer than $2^{|V|}$. Consequently,

$$|uxy| \leq 2^{|V|} < 2^{|V|+2} = p.$$

□

As with PPI and regular languages, we use the contrapositive of theorem 3.5 to prove that a language is not context-free. That requires assuming the negation of PPII as an hypothesis. The logical expression of this negation is,
$(\forall p \in \mathbb{N}, p > 0)\,(\exists s \in L)$, $|s| > p$ and $(\forall u, v, x, y, z \in \Sigma^*)$, $s = uvxyz$;

1. $(\exists i \geq 0)\; uv^ixy^iz \notin L$; or
2. $|vy| = 0$; or
3. $|vxy| > p$.

In most applications, items 2 and 3 are assumed to be false. This is a natural assumption, since $|vy| = 0$ means that $v = y = \varepsilon$; and it is always possible to find segments $v, x, y \in \Sigma^*$ of s, where $|vxy| \leq p$.

Next is an example of a non-context-free language

Example 3.7.1. The language $L = \{w \in \{a, b, c\}^* : w = a^nb^nc^n,\ n \in \mathbb{N}\}$, is not a context-free language. To verify that is the case, for each $p \in \mathbb{N}$, $p > 0$ we choose

$$s = a^pb^pc^p \in L.$$

Now, consider a segmentation of s, $s = uvxyz$, with $|vy| > 0$ and $|vxy| \leq p$. Since subsegment $|vxy| \leq p$, it cannot contain all three symbols a, b and c. Thus, uv^2xy^2z cannot longer have the same amount of a's, b's and c's.

Figure 20 updates the Venn diagram of language classification by adding context-free languages and the existence of a non-context-free language.

3.8 Set-theoretical Properties

Theorem 3.6 (Closure Properties)**.** *Let L_1 and L_2 be context-free languages, Then,*

1. *$L_1 \cup L_2$ is a context-free language;*
2. *L_1L_2 is a context-free language; and*
3. *L_1^* is a context-free language.*

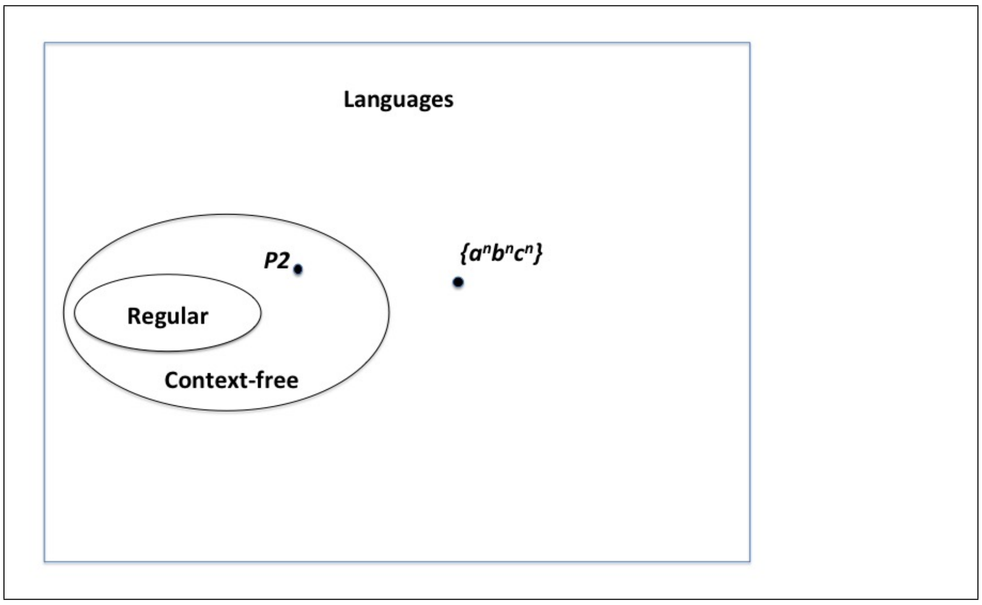

Figure 20: Classification of Languages.

Proof. Since L_1 and L_2 are context-free languages, there are context-free grammars $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. Let S be a new variable, $S \notin V_1 \cup V_2$.
To prove Claim 1, construct the grammar $G = (V, \Sigma, R, S)$, where

1. $V = V_1 \cup V_2 \cup \{S\}$, and
2. $R = R_1 \cup R_2 \cup \{S \to S_1 | S_2\}$.

Clearly, $L(G) = L_1 \cup L_2$ and therefore, $L_1 \cup L_2$ is a context-free language.
To prove Claim 2, construct the grammar $G = (V, \Sigma, R, S)$, where

1. $V = V_1 \cup V_2 \cup \{S\}$, and
2. $R = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$.

Clearly, $L(G) = L_1 L_2$ and therefore, $L_1 L_2$ is a context-free language. To prove Claim 3, construct the grammar $G = (V, \Sigma, R, S)$, where

1. $V = V_1 \cup V_2 \cup \{S\}$, and
2. $R = R_1 \cup R_2 \cup \{S \to S_1 S | \varepsilon\}$.

Clearly, $L(G) = L_1^*$ and therefore, L_1^* is a context-free language. $\square$

Theorem 3.7. *Let L be a context-free language and R be a regular language. Then, $L \cap R$ is a context-free language.*

Proof. Let $P = (Q_1, \Sigma, \Gamma, \delta_1, q_1, F_1)$ be a pushdown automaton that recognizes L and $F = (Q_2, \Sigma, \delta_2, q_2, F_2)$ a nondeterministic finite state automaton that recognizes R. Construct the pushdown automaton $T = (Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F_1 \times F_2)$ where for $a \in \Sigma \cup \{\varepsilon\}$,

$$\delta((q,p),a,\alpha) = \{((r,s),\beta) : (r,\beta) \in \delta_1(q,a,\alpha),\ s \in \delta_2(p,a)\}.$$

Since T accepts a string if and only if the computation ends with $((q,p),\varepsilon)$, with $q \in F_1$ and $p \in F_2$, the string is accepted by P and F, as well. Thus, $L(P) = L \cap R$, which proves that $L \cap R$ is a context-free language. $\square$

Chapter 4

Turing Machines

The problem of what is and what is not computable, or in the parlance of the time, what was *effectively calculable*, was posed way before the advent of modern electronic computers. In the year 1936, Alonso Church, a renowned USA logician and computer scientist, conjectured that

only recursive functions can be effectively calculable.

He arrived at this conjecture through the use of $\lambda-$calculus, a formal system in mathematical logic for expressing computations on the basis of mathematical functions and their variables. His statement, which was restricted to functions defined and with values in the natural numbers, is essentially what is nowadays known as *Church's thesis.* But by then, Kurt Gödel, an Austrian mathematician living in the United States and considered to be one of the most significant logicians in history, found Church's claim unconvincing. Accounts of the time said that his actual expression was *highly unsatisfactory*, and as a consequence, the problem remained open.

Unaware of Church's conjecture, Alan Turing, a British mathematician and founder of computer science, took a different approach. Instead of the mathematics of the problem, Turing observed the basic operations performed by a human computer during a computation, paying special attention to her/his mechanical moves. He concluded that the mechanics of a human computer was reduced to three basic operations namely, read, write and move across a page. He thought of the page as a linear memory device or *tape* and conceived a finite state automaton capable of executing sequences of read, write and move operations. This automaton was called *Turing machine*. Alan Turing conjectured that

Let Σ be an alphabet. A partial function defined and with values in Σ^* is effectively calculable if an only if it is computable by a Turing machine.

This conjecture became to be known as Turing's thesis. Just as Church's thesis, Turing's thesis is not a proved mathematical fact but a conjecture. However, Kurt Gödel found Turing's construction and arguments convincing enough, although later in his life, he raised a few questions on the nature of Turing machines. It turned out that in 1937 Turing himself proved that the concepts of λ-calculus definability, meaning what can be solved with λ-calculus, and Turing computability, meaning what can be solved with a Turing machine, were equivalent when applied to numeric functions defined and with values in the natural numbers. That give way to what is currently known as the *Turing-Church Thesis*.

Turing's thesis is nowadays supported by overwhelming evidence. Its success has led some computer scientists to claim that the Turing-Church Thesis is indeed the insistently sought-after definition of computer algorithm; meaning *computer algorithm* and *Turing machine* are synonyms. This statement is debatable, since there are some computational procedures that do not conform one hundred percent to the Turing model. Examples of them are distributed computations, like Web searches, or the operation of a multithreaded, multiuser operating system, among others. But, as we will discuss in this and the next chapters, the Turing machine model is the most powerful general description currently known of algorithms for solving decision problems, and serves as the basis for the construction of a theoretical body to study computability, meaning what can or can be not computed, and the time and space complexity of finding a solution.

4.1 Deterministic Turing Machine

Definition 4.1.1. A deterministic Turing machine (DTM) is a seven-tuple

$$T = (Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r),$$

where

1. Q is a finite set of state labels,
2. Σ is an alphabet, called language's alphabet,
3. Γ is also an alphabet, called tape's alphabet. This is defined so that

   $$\Sigma \subset \Gamma, \text{ and } \sqcup \in \Gamma - \Sigma;$$

   where $\sqcup$ is the blank symbol,
4. δ is the transition function,

   $$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

   Here L means move one symbol to the left, and R, move one symbol to the right;
5. $q_s, q_a, q_r \in Q$ are the start, accept and reject state, respectively. Here, $q_a \neq q_r$.

The tape is a string in Γ^*. A computation with a TM is best described in terms of strings in $\Gamma^* Q \Gamma^*$ called *configurations*.

Definition 4.1.2. A configuration is a string over $\Gamma^* Q \Gamma^*$,

$$C = \gamma_1 \cdots \gamma_k q \gamma_{k+1} \cdots \gamma_n,$$

where $\gamma_i \in \Gamma$ and $q \in Q$. This configuration represents the tape at state q. State q is inserted to the left of the next symbol to be read in the computation.

We say that configuration C_1 *yields* configuration C_2 if and only if C_2 represents the transformation of the tape from C_1 under the action of a TM transition. Thus, the yield

$$C_1 = \gamma_1 \cdots xqay \cdots \gamma_n \to C_2 = \gamma_1 \cdots rxby \cdots \gamma_n$$

corresponds to a transition of the form

$$\delta(q,a) = (r,b,L).$$

While the yield

$$C_1 = \gamma_1 \cdots xqay \cdots \gamma_n \to C_2 = \gamma_1 \cdots xbry \cdots \gamma_n$$

corresponds to

$$\delta(q,a) = (r,b,R).$$

In the case that $\delta(q,a) = \delta(r,b,L)$ and a is the leftmost symbol in the configuration, the transition just replaces a with b. If $\delta(q,a) = (r,b,R)$ and a is the rightmost symbol in the configuration, the transition performs the next yield

$$C_1 = \gamma_1 \cdots xqa \to C_2 = \gamma_1 \cdots xbr\sqcup.$$

Definition 4.1.3. A computation of a TM, on input $w = w_1 \cdots w_n$, is a sequence of configurations connected by yields, which starts with the configuration

$$C_0 = q_s w_1 \cdots w_n,$$

the so-called start configuration.

A TM accepts a string w and halts if and only if the computation reaches a configuration with state q_a, and rejects it, and halts, if it reaches a configuration with state q_r. States q_a and q_r are the sole *halting states*, meaning that the TM does not halt in any other state. If a TM never reaches q_a or q_r on an input, it is said to have entered an *infinite loop*, or simply, that it *loops* on that input.

The language of a TM T, denoted $L(T)$, is the set of all strings in Σ^* that the machine accepts.

The fact that a TM may loop is to be included when defining language recognition with a Turing machine.

Definition 4.1.4. A TM that does not loop on any input is called a *decider*. A language L over Σ is said to be *decidable* if there exists a TM T such that $L = L(T)$ and T is a decider. A language L over Σ is said to be *Turing-recognizable* it there exists a TM T such that $L = L(T)$ but T is not necessarily a decider.

Algorithm 4 A Decider of L

```
1: procedure D(On input $w \in \{a,b,c\}^*$)
2:     if $|w|$ is not a multiple of 3 then
3:         Reject.
4:     Read $w$ from left to right.
5:     if $a$ appears after $b$ or $c$, or $b$ after $c$ then
6:         Reject.
7:     while There are non-blank symbols in $w$ do
8:         Read $w$ from left to right replacing one $a$, one $b$ and one $c$ with $\sqcup$.
9:         if either $a$, $b$ or $c$ was not found but $w$ is not blank then
10:            Reject.
11:    Accept.
```

Remark 4.1.1. Notice that according to Definition 4.1.4, the set of all decidable languages is a subset of all Turing-recognizable languages.

Example 4.1.1. We claim that the procedure described in Algorithm 21 is a decider of the language $L = \{w \in \{a,b,c\}^* : w = a^n b^n c^n,\ n \in \mathbb{N}\}$. To sustain this claim it is necessary to make sure that the instructions in Algorithm 21 are executable by a Turing machine. For instance, statement 2 in the algorithm seems to be an "*intelligent*" decision. Can be executed by a TM? We verify that by outlining a general mathematical form of the corresponding transitions. To prove that statement 2 in Algorithm 21 is executable with a TM we set the state labels q_0, q_1 and q_2, let $w = w_0 w_1 \cdots w_{n-1}$ be the input string and let $< i >_3 = i \bmod 3$. Then, define the transitions

$$\begin{aligned}
&\delta(q_s, w_0) = (q_0, w_0, R);\ \text{and for } i = 1, ..., n-1,\\
&\delta(q_{<i-1>_3}, w_i) = (q_{<i>_3}, w_i, R)\\
&\delta(q_0, \sqcup) = \delta(q_1, \sqcup) = (q_r, \sqcup, L)\\
&\delta(q_2, \sqcup) = \delta(r, w_{n-1}, L);
\end{aligned}$$

where r is a new state. Clearly, this sequence of transitions reject if the length of w is not a multiple of 3, and starts a return towards the leftmost symbol in the tape, if it is. After verifying that D is a TM, by representing each statement as a sequence of TM transitions, if necessary; we observe that all possible cases of input string w are considered in the procedure, and on each input D either accepts or rejects. Thus, we conclude that $L = \{w \in \{a,b,c\}^* : w = a^n b^n c^n,\ n \in \mathbb{N}\}$ is a decidable language.

4.2 Multitape Turing Machines

Definition 4.2.1. A k-tape Turing machine (k-tape TM) is a DTM whose transitions are defined as

$$\begin{aligned}&\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L,R\}^k,\\ &\delta(q,(\gamma_1,...,\gamma_k)) = (r,(\beta_1,....,\beta_k),(X_1,...,X_k));\ X_i = L \text{ or } R.\end{aligned}$$

Thus, at each stage, the machine reads and write simultaneously on k tapes and moves one space to the right or to the left, independently on each tape.

A computation with a k-tape TM is a sequence of arrays of configurations $C^i = (C_1^{(i)},\dots,C_k^{(i)})$ that starts with

$$\begin{aligned}C^{(0)} &= (C_1^{(0)},\dots,C_k^{(0)}), \text{ where}\\ C_1^{(0)} &= q_s w_1 \cdots w_n,\\ C_2^{(0)} &= \cdots = C_k^{(0)} = q_s \sqcup .\end{aligned}$$

A computation accepts the input string w and halts, if and only if there is a an array $C^{(j)}$ that has a configuration with the state q_a, and rejects and halts, if there is an array $C^{(j)}$ with a configuration that has the state with q_r.

Theorem 4.1. *Each k-tape TM has an equivalent DTM.*

Proof. Let M be a k-tape TM. Construct a single tape TM T that reproduces the sequence of array configurations of M linearly, from left to right. This is, T is designed to produce the sequence of yields

$$C_1^{(0)},C_2^{(0)},...,C_k^{(0)},C_1^{(1)},...,C_i^{(j)},....$$

Thus, T accepts whenever M accepts, and rejects, whenever M rejects. So, $L(T) = L(M)$. □

Example 4.2.1. The 2-tape Turing machine described in Algorithm 5 decides $L = \{w \in \{a,b,c\}^* : w = a^n b^n c^n,\ n \in \mathbb{N}\}$. The machine keeps the input string in the first tape and uses the second to compare the amounts of a's, b's and c's.

4.3 Nondeterministic Turing Machines

Definition 4.3.1. A Turing machine $(Q,\Sigma,\Gamma,\delta,q_s,q_a,q_r)$ is nondeterministic (NTM) if its transition is defined by the relation

$$\begin{aligned}&\delta : Q \times \Gamma \to \mathscr{P}(Q \times \Gamma \times \{L,R\});\\ &\delta(q,a) = \{(r,b,X) : \text{for some } r \in Q,\ b \in \Sigma,\ X \in \{L,R\}\}.\end{aligned}$$

A computation with a NTM is a directed tree representing rooted in the initial configuration $q_s w_1 \cdots w_n$, whose edges represent yields and whose nodes are the yielded configurations.

Algorithm 5 A 2-tape Decider of L

```
procedure M(On input $w \in \{a,b,c\}^*$)
    Write $w$ on tape 1.
    Read $w$ from left to right.
    if $a$ appears after $b$ or $c$, or $b$ after $c$ then
        Reject.
    From left to right, read the $a$'s in $w$ copying each read $a$ in tape 2.
    After the last $a$ is read, insert the symbol $.
    while The start of tape 2 has not been reached do
        Read each $b$'s in $w$, deleting an $a$ for each $b$ and moving to the left.
    if The start of tape 2 was not reached, or there are unread $b$'s then
        Reject.
    while Symbol $ is not encountered in tape 2 do
        Read the $c$'s copying each read $c$ in tape 2.
    if Symbol $ was not encountered, or there are unread $c$'s then
        Reject.
    Accept.
```

Remark 4.3.1. As with previous finite state machine models, nondeterministic Turing machines convey the idea of a search for a solution. It is worth noticing that since TMs can change states reading and writing the same symbol, which is equivalent to not reading a symbol, NTMs do not need ε transitions.

Definition 4.3.2. A string is accepted by an NTM if and only if one of the branches in the computation tree reaches a configuration with state q_a; and rejects it, if all the branches in its computation tree end in a configuration with state q_r.

Example 4.3.1. Algorithm 6 is a nondeterministic decider of $L = \{w \in \{a,b,c\}^* : w = a^n b^n c^n,\ n \in \mathbb{N}\}$. If $w \in L$, then one of the nondeterministic choices in instruction 2 will start an accepting computation branch. And if $w \notin L$, all computation branches reject.

Theorem 4.2. *Each nondeterministic Turing machine has an equivalent deterministic Turing machine.*

Proof. Let N be a nondeterministic Turing machine. We will construct an equivalent 3-tape deterministic Turing machine D. Tape 1 keeps the input string, tape 2 performs the computation of a branch in N, tape 3 contains the pointers to legal computation branches that have not reached a reject state. Each pointer is a string over the alphabet $\{1,2,...,m\}$ where m is the size of the largest $\delta(q,\alpha)$ set in the transitions of N. Each index $1d_1d_2...d_k$ represents a segment of a computation branch starting from the root 1, its child d_1, the child of d_1, d_2, and so on. D produces the tree of configurations of the computation of w, breadth first. Algorithm 7 is a high-level description of Turing machine D. $\square$

Algorithm 6 A Nondeterministic Decider of L

```
procedure ND(On input $w \in \{a,b,c\}^*$)
    Nondeterministically split $w = uvz$
    if $u$ has no $a$, or any other character besides $a$ then
        Reject.
    else if $v$ has no $b$, or any other character besides $b$ then
        Reject.
    else if $z$ has no $c$, or any other character besides $c$ then
        Reject.
    if $|u| = |v| = |z|$ then
        Accept
    else
        Reject.
```

Remark 4.3.2. It is worth remarking that in the theory of pushdown automaton there are no equivalents to Theorem 2.4 of Chapter 2, or Theorem 4.2 of Chapter 3. It turns out that no such theorem exists. This is, nondeterministic pushdown automaton do not have an equivalent deterministic pushdown automaton. Indeed, the proof techniques used in the proofs of theorems 2.4 and 4.2 are not valid in the context of pushdown automata. In particular, the method production of all nondeterministic computations level by level of the computation tree used in the proof of Theorem 4.2 is not possible with a single stack memory. Also, the interpretation of all states in a level of the computation tree as a set as a state used in the proof of Theorem 2.4, is not reproducible in a pushdown automata as the stacks in a level are not necessarily the same, precluding thus the use of a set of states as a single state, along with a single stack.

4.4 Enumerators

Enumerators are similar to 2-tape Turing machines. Their main difference is that an enumerator is not designed to accept or reject a string but to produce a list of strings. Consequently, enumerators do not have a reject state, and its accept state means that the list is complete.

Definition 4.4.1. An enumerator is a seven-tuple $E = (Q,\Sigma,\Gamma,\delta,q_s,q_p,q_a)$ where Q,Σ, and Γ are as before, $q_s \in Q$ is the start state, $q_a \in Q$ is the accept state and $q_p \in Q$ is the *print* state. Each enumerator has a *working tape* and an *output tape*. The working tape is the usual read and write Turing machine tape. The output tape is a write only tape that stores strings in Σ^*, eventually separated by $\sqcup$. When an enumerator enters the print state, it outputs the contents of the output tape.

Algorithm 7 Deterministic simulation of a non-deterministic TM

```
1: procedure $M$(On input $< N, w, m >$)
2:     Copy $w$ on tape 1.
3:     Initialize tape 3 with 1.
4:     while $q_a$ has not been reached and tape 3 is not blank do
5:         while There is an address not updated do
6:             Select next address.
7:             for $d = 1 : m$ do
8:                 Concatenate address and $d$.
9:                 if New address is valid then
10:                    Simulate the computation branch on tape 2.
11:                    if $q_r$ is reached then
12:                        Eliminate the address.
13:    if $q_a$ is reached then
14:        Accept.
15:    if Tape 3 is blank then
16:        Reject.
```

Algorithm 8 Simple Enumerator

```
1: procedure $E$(On input $< A, m >$, $m \in \mathbb{N}$ )
2:     for $w \in \Sigma^*$, $|w| \leq m$ do
3:         Run $A$ on $w$
4:         if $A$ accepts then
5:             Write $w$ on the output tape
6:             Write $\sqcup$
7:     Accept.
8:     Print output tape.
```

An enumerator's transition map is defined as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \times \Gamma$$
$$\delta(q, \gamma) = (r, \beta, X, \alpha), \text{ where } X = L \text{ or } R.$$

Definition 4.4.2. A computation with an enumerator is similar to a computation with a 2-tape Turing machine. Initially the output tape is blank. A transition

$$\delta(q, \gamma) = (r, \beta, X, \alpha),\ X = L \text{ or } R;$$

instructs the automaton to read γ at state q, it overwrites γ with β, and writes a in the output tape. Upon completion of these operations, the automaton is in state r.

Example 4.4.1. Algorithm 8 is an enumerator all strings of length less than or equal to m in the language of a Finite State Automaton A; where m is a preselected natural number.

Algorithm 9 Recognizer for Enumerated Language

```
procedure $M_E$(On input w )
    Run $E$
    if $E$ writes $u$ on its output tape then
        if $u = w$ then
            Accept.
```

Algorithm 10 Enumeration of Turing-recognizable Language

```
procedure $E_M$(On input x )
    for $w \in \Sigma^*$ produced in lexicographic order do
        Run M on $w$.
        if $M$ accepts then
            Write $w$ on output tape.
```

Definition 4.4.3 (Recursively Enumerable Language)**.** A language L over an alphabet Σ is said to be *recursively enumerable* if there is an enumerator E that can lists its strings.

Remark 4.4.1. Notice that if a language is infinite, the enumerator will not halt. Therefore, just as the statement that a language is recognizable by a Turing machine, the statement that a language is recursively enumerable is essentially, a mathematical statement.

Example 4.4.2. By removing the condition of printing only the strings of length less than or equal to m in Algorithm 8 we get a proof that a regular language is recursively enumerable.

Theorem 4.3. *A language is recursively enumerable if and only if is Turing recognizable.*

Proof. Assume first that a language L is recursively enumerable. Then, there is an enumerator E that lists its strings. Then, the Turing machine M_E described in Algorithm 9 recognizes L.

Assume now that L is Turing-recognizable and that M is a recognizer. Algorithm 10 describes an enumerator for L. □

Remark 4.4.2. If L is infinite, E_M will not halt. Now, if language L is recognizable but not decidable, E_M will stop writing strings on its output tape whenever M falls in a loop. That will not happen if L is decidable. This property of allowing its strings to be written uninterruptedly is conveyed by saying that L is *enumerable*. Thus, enumerable language is synonym of decidable language, and recursively enumerable language is synonym of Turing-recognizable language.

Chapter 5

Decidability

According to Definition 21 in Chapter 4, a language is decidable if it is the language of a Decider, this is, a Turing Machine that either accepts or rejects an input, but never loops. Thus, *decidable language* is synonym of problem *algorithmically solvable*. This chapter discusses some examples of such problems and the most relevant theoretical aspects of decidability, *i.e. solvability*, including its limits.

5.1 Some Decidable Languages

The question of whether a formal language is algorithmically decidable is a crucial first step in software design endeavors, where the questions of

1. Given an algorithm and an input string, will the algorithm accept the string as a solution?

2. Is the set of accepted strings of a given algorithm, nonempty?

3. Are two different algorithms, equivalent?

These are all *testing* problems and the field of software engineering will greatly benefit from algorithmic answers to these questions. Next are discussions on the algorithmic solvability of these problems when the algorithms are representable as Finite State Automata, and Pushdown automata.

Example 5.1.1. Let

$$A_{DFA} = \{< B, w >: \ B \text{ is a FSA that accepts } w\}.$$

Formal language A_{DFA} encodes the problem of *testing* whether a string is a member of language $L(B)$. This kind of test is common in situations in which the language L has been specified independently of automaton B, and the question is indeed, Is $L = L(B)$?

The ability to solve A_{DFA} algorithmically ensures the validity of the test.

Algorithm 11 A Decider of A_{DFA}

```
procedure D(On input < B, w >)
    Run B on w.
    if B end in an accepting state then
        Accept.
    else
        Reject.
```

Algorithm 12 A Decider of E_{DFA}

```
procedure T (On input < B >)
    Mark the start state of B.
    while There is an unmarked state connected to a marked state do
        Mark the unmarked state.
    if No accept state of B is marked then
        Accept.
    else
        Reject.
```

Theorem 5.1. *A_{DFA} is decidable.*

Proof. The Turing machine described in Algorithm 11 is a decider of A_{DFA}. □

Example 5.1.2. A similar test is usually conducted for regular expressions. Let,

$$A_{REX} = \{< R, w >: \text{ } R \text{ is a regular expression and } w \text{ a string generated by } R\}.$$

Theorem 5.2. *A_{REX} is decidable.*

Proof. Find the FSA associated to R and use the decider described in Algorithm 11. □

Assume that for a given FSA B, $D(< B, w >)$ is rejected for all the tested w. In such a case, a natural question to ask is whether $L(B) = \emptyset$. This is the subject of the next example.

Example 5.1.3. Let

$$E_{DFA} = \{< B >: \text{ } B \text{ is a FSA and } L(B) = \emptyset\}.$$

Theorem 5.3. *E_{DFA} is decidable.*

Proof. The Turing machine described in Algorithm 12 decides E_{DFA}. □

Next is the problem of testing whether two FSA are equivalent.

Algorithm 13 Recognizing the symmetric difference of $L(A)$ and $L(B)$

```
procedure $C_{A,B}$ (On input $w$)
    Run $A$ on $w$.
    if $A$ accepts then
        Run $B$ on $w$.
        if $B$ accepts then
            Reject.
        else
            Accept.
    else
        Run $B$ on $w$
        if $B$ accepts then
            Acccept.
        else
            Reject.
```

Example 5.1.4. Let

$$EQ_{DFA} = \{< A,B >: \text{ } A \text{ and } B \text{ are FSA and } L(A) = L(B)\}.$$

Theorem 5.4. *EQ_{DFA} is decidable.*

Proof. First construct the Turing machine $C_{A,B}$ described in Algorithm 13. $C_{A,B}$ accepts only strings that are in the symmetric difference of $L(A)$ and $L(B)$, this is

$$S = (L(A) - L(B)) \cup (L(B) - L(A)).$$

Here $S - R = S \cap \bar{R}$. Since $L(A) = L(B)$ if and only if $S = \emptyset$, the strategy is to test $C_{A,B}$ with Algorithm 12. □

Let's consider now similar problems but with context-free grammars instead of Finite State Automata.

Example 5.1.5. Let

$$A_{CFG} = \{< G,w >: \text{ } G \text{ is a CFG that generates } w\}.$$

Theorem 5.5. *A_{CFG} is decidable.*

Proof. The Turing machine S described in Algorithm 14 decides A_{CFG}. □

Corollary 5.1. *Any context-free language is decidable.*

Proof. Let L be a context-free language and G be the context-free grammar that generates L. Then, for each $w \in \Sigma^*$, $w \in L$ if and only if $S(< G,w >)$ accepts. Thus, L is decided by $S(< G, >)$. □

Algorithm 14 Deciding A_{CFG}

```
1: procedure S (On input $< G, w >$)
2:     Transform $G$ into Chomsky normal form.
3:     Compute $n \leftarrow |w|$.
4:     Enumerate all $2n-1$-step derivations of $G$ in CNF.
5:     for Each $2n-1$-step derivation do
6:         if $w$ is generated then
7:             Accept.
8:     Reject.
```

Algorithm 15 Deciding E_{CFG}

```
1: procedure R (On input $< G >$)
2:     Transform $G$ into Chomsky Normal Form.
3:     Mark all variables $A$ for which there is a rule $A \to a$, $a$ terminal.
4:     while $A \to BC$ is a rule and $B$ and $C$ are marked do
5:         Mark $A$.
6:     if Start variable is marked then
7:         Accept.
8:     else
9:         Reject.
```

Finally, we examine the problem of deciding whether a context-free grammar generates no strings.

Example 5.1.6. Let

$$E_{CFG} = \{< G >\colon\ G \text{ is a CFG and } L(G) = \emptyset\}.$$

Theorem 5.6. *E_{CFG} is decidable.*

Proof. The Turing machine R that decides E_{CFG}, which is described in Algorithm 15, uses the same exhaustive search and mark technique already used in the construction of T, the decider of E_{DFA} described in Algorithm 12. □

Remark 5.1.1. So far, we have been successful in extending results for Finite State Automata to Context-free grammars. However, the strategy used to decide whether two Finite State Automata were equivalent, this is the construction of $C_{A,B}$ in Algorithm 13 for deciding the symmetric difference of the languages of A and B, is not extensible to context-free grammars, simply because context-free grammars are not closed under intersections of complementations. The problem of whether

$$EQ_{CFG} = \{< G, H >\colon\ G \text{ and } H \text{ CFG and } L(G) = L(H)\},$$

cannot be answered yet.

5.2 The Acceptance Problem for Turing Machines

Decidability is further complicated when when we consider tests that involve Turing machines instead of context-free grammars. The most basic test problem, which turns out to be a common problem in software engineering, is the so-called *Acceptance Problem*. This is stated as follows,

$$A_{TM} = \{< M, w >: \ M \text{ is a TM and accepts } w\}.$$

In natural language, the Acceptance Problem is stated as: "*Given a Turing machine M and a string w, Does M accept w?*" . The main difficulty in finding an algorithmic solution for A_{TM} is the existence of recognizers that are not deciders. Indeed, we can easily prove that

Theorem 5.7. *A_{TM} is Turing-recognizable.*

Proof. A recognizer for A_{TM} is a machine that on input $< M, w >$ runs M on w and *accepts* if M accepts. □

However, an attempt to prove that A_{TM} is decidable leads to a contradiction.

Theorem 5.8. *A_{TM} is not decidable.*

Proof. By contradiction. Assume that A_{TM} is decidable. Then, there is a decider H such that

$$H(< M, w >) = \begin{cases} \text{accept if } M \text{ accepts;} \\ \text{reject if } M \text{ does not accept.} \end{cases}$$

In particular, H decides on input $< M, < M >>$, meaning the input pair formed with Turing machine M and $< M >$, the string describing it. By restricting the use of H to inputs of the kind of $< M, < M >>$, we get a decider

$$\bar{H}(< M >) = \begin{cases} \text{accept if } M \text{ accepts } < M >; \\ \text{reject if } M \text{ does not accept } < M > . \end{cases}$$

Now, by exchanging accept and reject states in $\bar{H}$, we get a complementary decider D, which

$$D(< M >) = \begin{cases} \text{reject if } M \text{ accepts } < M >; \\ \text{accept if } M \text{ does not accept } < M > . \end{cases}$$

The contradiction arises when D is applied to $< D >$. In such case, we have that

$$D(< D >) \text{ reject if } D \text{ accepts } < D >;$$

which is not possible unless the *accept* and *reject* state of D are the same. □

Corollary 5.1 and Theorem 5.8 allow the extension of the taxonomy of formal languages that is shown in Figure 21

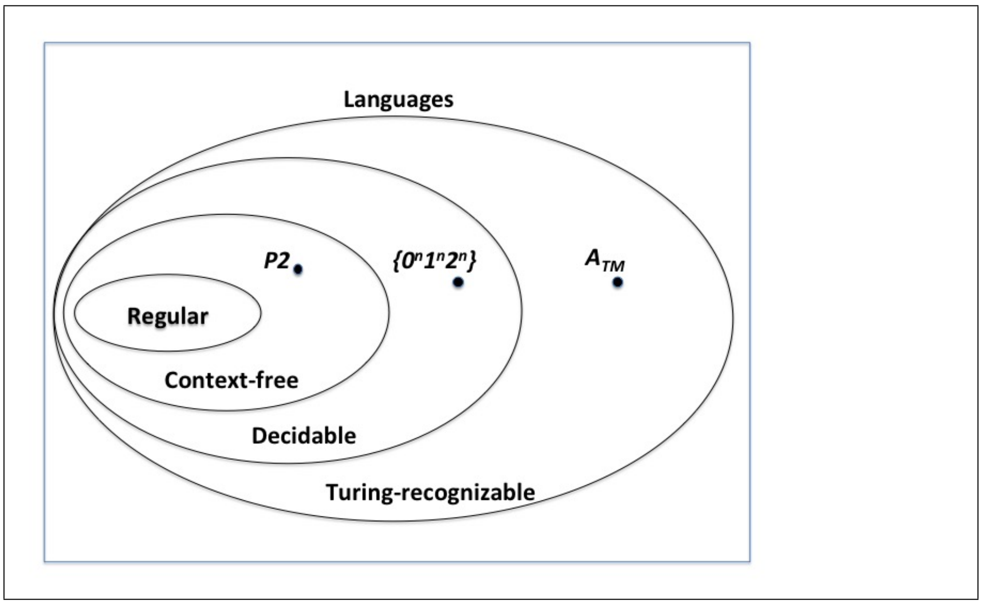

Figure 21: Taxonomy of formal languages

5.3 Existence of Turing-unrecognizable Languages

Undecidable, but Turing-recognizable languages are encodings of problems that do not have a full fledged algorithmic solution, but whose correct instances can be verified automatically. A language is *Turing-unrecognizable* if that is not longer the case. The main theorem in this section proves there are uncountably many such languages. This central theorem is proved with a rather simple set countability argument, which we discussed next.

Lemma 5.1. *Let Σ be an alphabet. Then Σ^* is infinite countable.*

Proof. By definition, Σ^* is the union of the collection of finite sets $\{\Sigma^n : n \in \mathbb{N}\}$. Since an infinite countable union of finite set results in a countable set, Σ^* is infinite countable. □

Corollary 5.2. *The set $\mathscr{T}$ of all Turing machines over an alphabet Σ is infinite countable.*

Proof. Each Turing machine is completely described by a seven-tuple

$$(Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r)$$

of finite objects, and thus, it can be encoded as a string in Σ^*. Therefore, $\mathscr{T} \subset \Sigma^*$ and since Σ^* is infinite countable, so is $\mathscr{T}$. □

Lemma 5.2. *Let Σ be an alphabet, with $|\Sigma| \geq 2$. Then, the set of infinite sequences*

$$S_\Sigma = \{s : s = (a_n),\ a_n \in \Sigma,\ n \in \mathbb{N}\},$$

is infinite but not countable.

Proof. By contradiction. Let's assume that S is countable. Then, the sequences in S can be enumerated so that

$$S_\Sigma = \{s^{(i)} : s^{(i)} = (a_n^{(i)}),\ a_n^{(i)} \in \Sigma,\ i,n \in \mathbb{N}\}.$$

Since $|\Sigma| \geq 2$, we can construct a sequence $\sigma = (\sigma_i)$ such that

$$\forall i \in \mathbb{N},\ \sigma_i \neq a_i^{(i)}.$$

But then, $\sigma \neq s^{(i)}$, $\forall i \in \mathbb{N}$. Thus, $\sigma \notin S_\Sigma$ which is a contradiction, because σ is an infinite sequence over Σ. □

Theorem 5.9 (Existence of Turing-unrecognizable Languages)**.** *The set*

$$\mathscr{U} = \{L : L \textit{ is a Turing} - \textit{unrecognizable language over } \Sigma\}$$

is infinite, not countable.

Proof. Let $\Sigma = \{0,1\}$. Since Σ^* is infinite countable, we enumerate

$$\Sigma^* = \{s^{(i)} : s^{(i)} = (a_n^{(i)}),\ a_n^{(i)} \in \{0,1\},\ i,n \in \mathbb{N}\};$$

and for each language L over Σ, define the characteristic sequence $z^{(L)}$ as

$$z_i^{(L)} = \begin{cases} 1 & \text{if } s^{(i)} \in L \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, for each language L, there is a unique infinite sequence $z^{(L)} \in S_\Sigma$. Reciprocally, each non-null infinite sequence $z = (z_i)$ over $\{0,1\}$ determines a unique language

$$L_z = \{s^{(i)} \in \Sigma^* : s^{(i)} \text{such that } z_i = 1\}.$$

This shows that there is a one-to-one correspondence between the set $\mathscr{P}(\Sigma^*)$ of all languages over Σ and S_Σ. Consequently $\mathscr{P}(\Sigma^*)$ is infinite uncountable. Since

$$\mathscr{U} = \mathscr{P}(\Sigma^*) - \mathscr{T}$$

and the difference of an infinite uncountable and an infinite countable set is an infinite uncountable set, so is $\mathscr{U}$. □

Remark 5.3.1. Theorem 5.9 completes the classification of languages over an alphabet Σ, in terms of their automatic solvability as decision problems. Figure 22 is the Venn diagram of the classification.

The fact that there are uncountable infinite problems whose instances cannot even be verified automatically may be counterintuitive. After all, computers seems to have an endless power to solve problems. A couple of remarks are in order here,

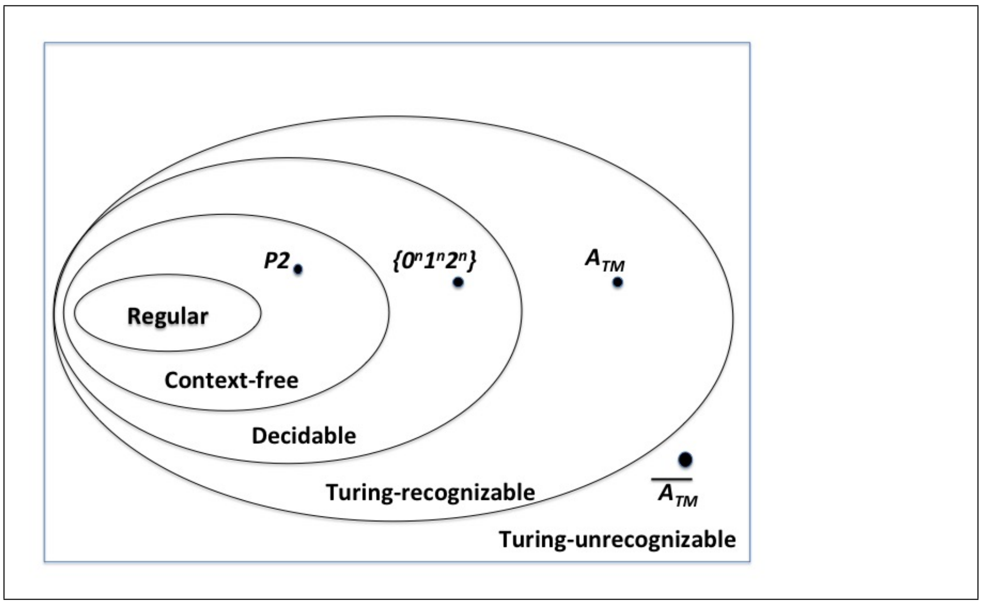

Figure 22: Complete Venn diagram of algorithmically solvable, verifiable and unsolvable problems.

1. Recall that solvable or verifiable problems are couched in the language of logic. Their strings are essentially those that make a well-formed predicate true, and the Turing machine that recognizes the solution is thus, structured around that predicate's logic and its properties;

2. If a set of strings is produced randomly, or what is the case in some applications, no complete logical description of the problem is available, the task of finding a decider of recognizer of a member in that set of strings is essentially unsolvable (except perhaps via a lucky strike of pure inspiration).

Definition 5.3.1. A language L over an alphabet Σ is said to be co-Turing recognizable if its complement $\bar{L}$ is Turing-recognizable.

Theorem 5.10. *A language is decidable if and only if it is both, Turing and co-Turing recognizable.*

Proof. Assume first that L is a decidable language. Then, by definition of Turing-recognizable, L is Turing-recognizable. Since the complement of a decidable language is decidable $\bar{L}$ is also decidable and Turing recognizable.

Assume now that L is Turing and co-Turing recognizable. Then, there are M_L and $M_{\bar{L}}$, recognizers of L and $\bar{L}$, respectively. We construct the 2-tape Turing machine described in Algorithm 16.

M decides L because, for each $w \in \Sigma^*$, either $w \in L$ or $w \in \bar{L}$. $\square$

Algorithm 16 Decider for Turing and co-Turing recognizable languages

```
1: procedure $M$ (On input $w$)
2:     Run $M_L$ and $M_{\bar{L}}$ in parallel.
3:     if $M_L$ accepts then
4:         Accept.
5:     else if $M_{\bar{L}}$ accepts then
6:         Reject.
```

Corollary 5.3. *The language*

$$\overline{A_{TM}} = \{< M, w >: \ M \text{ Turing machine}, w \text{ string and } M \text{ does not accept } w\},$$

is not Turing-recognizable.

Proof. We proved that A_{TM} is Turing-recognizable. If $\overline{A_{TM}}$ were Turing-recognizable, then, because of Theorem 5.10, A_{TM} will be decidable. But Theorem 5.8 shows that that is not the case. Therefore, $\overline{A_{TM}}$ is not Turing-recognizable. □

Remark 5.3.2. Corollay 5.3 allows a negative refinement of item 1 in Remark 5.3.1. Language $\overline{A_{TM}}$ is described with a well-formed predicate, namely

$$P(M, w) =: \text{“}M \text{ Turing machine}, w \text{ string and } M \text{ does not accept } w.\text{''}$$

But in this case, the predicate does not yield a clue for the construction of a recognizer for the language. In conclusion, not every language that is described with a predicate is algorithmically treatable, either.

Chapter 6

Reducibility and Computable Functions

As mentioned in the brief introductory remark in Chapter 4, computable functions were the central object of study of the founders of Computer Science. The definition of computable function that is discussed and applied in this Chapter, is closely related to the one introduced by Alan Turing. This Chapter concentrates on the application of computable functions to mapping reducibility, which is basically, a mathematical description for the reduction of a problem (*i.e* formal language) to another one.

6.1 Reduction

The idea of *problem reduction* is common in Computer Science and in Mathematics. It essentially conveys the idea of solving a complex problem in terms of a simpler one, or in terms of one whose solution is already known. For instance, the problem of finding the area of a square is reduced to computing the square of the length of a side. A reduction of problems encoded in formal languages is described as follows,

Definition 6.1.1. A *reduction* of a language L to a language K is a function

$$\begin{aligned}&f : \Sigma^* \to \Sigma^*,\\&(\forall w \in \Sigma^*),\ w \in L \leftrightarrow f(w) \in K.\end{aligned}$$

Remark 6.1.1. It is worth remarking that $(\forall w \in \Sigma^*)$,

$$\begin{aligned}w \in L \leftrightarrow f(w) \in K &\longleftrightarrow (w \in L \to f(w) \in K) \wedge (f(w) \in K \to w \in L)\\&\longleftrightarrow (w \in L \to f(w) \in K) \wedge (w \notin L \to f(w) \notin K).\end{aligned}$$

Algorithm 17 Turing machine M'

```
1: procedure $M'$(On input $x$ )
2:     Run $M$ on $x$.
3:     if $M$ accepts then
4:         Accept.
5:     if $M$ rejects then
6:         Enter a loop.
```

Example 6.1.1. Let $L = \{w \in \{a,b,c\}^* : w = a^n b^n c^n,\ n \in \mathbb{N}\}$ and $K = \{w \in \{a,b,c\}^* : w = a^n b^n\}$. Then, the function

$$f : \{a,b,c\}^* \to \{a,b,c\}^*$$
$$f(w) = \begin{cases} a^n b^n & \text{if } w = a^n b^n c^n \\ ab^2 & \text{otherwise,} \end{cases}$$

is a reduction of L to K. Indeed, it is clear that if $w = a^n b^n c^n \in L$, then $f(w) = a^n b^n \in K$; and, using the contrapositive statement, if $w \notin L$ then $f(w) = ab^2 \notin K$.

Example 6.1.2. We define the Halting Problem as

$HALT = \{<M,w>: M$ is a Turing machine, w is a string and M halts on input $w\}$.

The next function is a reduction of A_{TM} to $HALT$,

$$f : \Sigma^* \to \Sigma^*$$
$$f(<M,w>) = <M',w>;$$

where M' is the Turing machine defined in Algorithm 17. To prove that f reduces A_{TM} to $HALT$ consider first $<M,w> \in A_{TM}$. Then, M accepts w and thus, $<M',w> \in HALT$. On the other hand, if $<M,w> \notin A_{TM}$, M either rejects w, or enters a loop on input w. If M rejects w, M' enters a loop, and thus, $<M',w> \notin HALT$. And if M loops on input w, as M is a subroutine inside M', M' also loops. So, in that case, $<M',w> \notin HALT$, also.

Example 6.1.3. The next mapping f reduces the language E_{TM} to EQ_{TM}.

$$f : \Sigma^* \to \Sigma^*$$
$$f(<M>) = <M,M_1>;$$

where M_1 is the Turing machine that rejects all its inputs. To prove that f is a reduction of E_{TM} to EQ_{TM}, take first $<M> \in E_{TM}$. Then, $L(M) = \emptyset$. Since $L(M_1) = \emptyset$ by construction, $L(M) = L(M_1)$. Thus, $<M,M_1> \in EQ_{TM}$. Reciprocally, if $<M> \notin E_{TM}$, then $L(M) \neq \emptyset$ and $L(M) \neq L(M_1)$. Thus, $<M,M_1> \notin EQ_{TM}$.

Remark 6.1.2. The image $f(L)$ of a reduction f of a language L to a language K does not necessarily cover the whole language K. In the previous examples only the image in Example 6.1.1 covers the whole language K. Thus, in general, $f(L)$ is a sublanguage of K.

Furthermore, if $f : L \to K$ is one-to-one then, the inverse f^{-1} of f is a reduction from $f(K)$ to L.

Theorem 6.1. *If L is reducible to K, then $\overline{L}$ is reducible to $\overline{K}$.*

Proof. Let f be a reduction of L to K. Then, according with Remark 6.1.1,

$$\begin{aligned} w \in L \leftrightarrow f(w) \in K &\longleftrightarrow (w \in L \to f(w) \in K) \wedge (w \notin L \to f(w) \notin K) \\ &\longleftrightarrow (f(w) \notin K \to w \notin L) \wedge (w \notin L \to f(w) \notin K) \\ &\longleftrightarrow w \notin L \leftrightarrow f(w) \notin K. \end{aligned}$$

Thus, $\overline{L}$ is reducible to $\overline{K}$ with the same function f. □

6.2 Computable Functions

Intuitively, a computable function is one whose values can be computed with a Turing machine. This idea is formalized in the next defintion.

Definition 6.2.1. A function $f : \Sigma^* \to \Sigma^*$ is said to be *computable*, if there is a Turing machine M which on every input $w \in \Sigma^*$, halts with just $f(w)$ on its tape.

Remark 6.2.1. Obviously, everything that an algorithm outputs on a given input w, corresponds to a computable function $f(w)$, although f may not be describable in a closed formula, as we have done in Examples 6.1.1, 6.1.2 and 6.1.3. If necessary, the tape can always be cleared to end with just $f(w)$. Notice that the machine may accept or reject w.

Example 6.2.1. The functions defined in Examples 6.1.1, 6.1.2 and 6.1.3 are computable, as demonstrated by the Turing machines described in Algorithm 18, 19, and 20. It is worth remarking that the constructions of M' in Algorithm 19, and M_1 in Algorithm 20 are both doable with the input provided. It is also important to emphasize that neither M' nor M_1 run as subroutines within F' and F_1, respectively. F' and F_1 just produce the string descriptions of these Turing machines.

Definition 6.2.2. A Language L is said to be *Turing-reducible* to a language K, denote $L \leq_m K$, if their reduction map is a computable function.

Example 6.2.2. The previous discussion shows that each of the reductions in Examples 6.1.1, 6.1.2 and 6.1.3 is a Turing-reduction.

Theorem 6.2. *If $L \leq_m K$, then $\overline{L} \leq_m \overline{K}$,*

Proof. Is a direct consequence of Theorem 6.1. □

Algorithm 18 Computing function in Example 6.1.1

```
1: procedure $M$(On input $w$ )
2:     if $w = a^n b^n c^n$ then
3:         Write $a^n b^n$ on tape.
4:         Halt.
5:     else
6:         Write $ab^2$ on tape.
7:         Halt.
```

Algorithm 19 Computing function in Example 6.1.2

```
1: procedure $F'$(On input $< M, w >$ )
2:     Construct $M'$.
3:     Write $< M', w >$ on tape.
4:     Halt.
```

6.3 Turing-reducibility and Decidability

Turing-reductions are an important tools for classifying languages into the decidable or undecidable categories. The next theorems state the principles used for this purpose.

Theorem 6.3. *If $L \leq_m K$ and K is decidable, then L is decidable.*

Proof. Let f be a computable reduction of L to K, and D be a decider of K. Then, Algorithm 21 is a decider of L. Indeed, since f is a reduction, $w \in L \leftrightarrow f(w) \in K$. And since f is computable, step 2 in Algorithm 21 is executable. Since D decides L, D accepts $f(w)$ if and only if $w \in L$. Therefore, N is a decider of L. □

Corollary 6.1. *If $L \leq_m K$ and L is undecidable, then K is undecidable.*

Proof. The contrapositive of the statement is

"If K is decidable, then $L \not\leq_m K$ or L decidable."

Since we consider only cases where $L \not\leq_m K$, $L \not\leq_m K$ is false. Then, by Theorem 6.3, L is decidbale. This makes the contrapositive of the statement true. □

Theorem 6.4. *HALT is undecidable.*

Proof. $A_{TM} \leq_m HALT$. □

Theorem 6.5. *If $L \leq_m K$ and K is Turing-recognizable, L is Turing-recognizable.*

Proof. Same as the proof of Theorem 6.3 but with D and N recognizers instead of deciders. □

Corollary 6.2. *If $L \leq_m K$ and L is not Turing-recognizable, then K is not Turing-recognizable.*

Algorithm 20 Computing function in Example 6.1.3

```
1: procedure $F_1$(On input $< M >$ )
2:     Construct $M_1$.
3:     Write $< M, M_1 >$ on tape.
4:     Halt.
```

Algorithm 21 Decider of reducible language

```
1: procedure $N$(On input $w$ )
2:     Compute $f(w)$.
3:     Run $D$ on w.
4:     if $D$ accepts then
5:         Accept.
6:     else
7:         Reject.
```

Proof. By contraposition. The contrapositive of the statement is "If K is Turing-recognizable, then $L \not\leq_m K$ or L is Turing-recognizable," which is true because of Theorem 6.5. □

Theorem 6.6. *EQ_{TM} is not Turing-recognizable.*

Proof. We use the fact that $\overline{A_{TM}}$ is not Turing-recognizable along with a Turing-reduction of $\overline{A_{TM}}$ to EQ_{TM}. The next function,

$$f : \Sigma^* \to \Sigma^*$$
$$f(< M, w >) = < M_1, T_{<M.w>} >$$

where M_1 is the Turing machine that rejects all inputs and $T_{<M,w>}$ is defined as in Algorithm 27.

Function f is a reduction, since if $< M, w > \notin A_{TM}$, then M does not accept w and therefore, $L(M_1) = L(T_{<M,w>}) = \emptyset$. If, on the other hand, $< M, w > \in A_{TM}$, M accepts w and $L(T_{<M,w>}) = \Sigma^* \neq L(M_1) = \emptyset$.

The Turing machine described in Algorithm 23 demonstrate that f is computable. Therefore, $\overline{A_{TM}}$ is Turing-reducible to EQ_{TM}. Thus, the Theorem is a consequence of Theorem 6.5.

□

Theorem 6.7. *$\overline{HALT}$ is not Turing-recognizable.*

Proof. By using the fact that $A_{TM} \leq_m HALT$ and Theorem 6.2, we have that $\overline{A_{TM}} \leq_m \overline{HALT}$. But since $\overline{A_{TM}}$ is not Turing-recognizable, so is $\overline{HALT}$. □

Theorem 6.8. *$\overline{EQ_{TM}}$ is undecidable.*

Algorithm 22 Turing machine $T_{<M,w>}$

```
1: procedure $T_{<M,w>}$ (On input $x$ )
2:     Run $M$ on $w$.
3:     if $M$ accepts then
4:         Accept.
```

Algorithm 23 Computing mapping f in Theorem 6.6

```
1: procedure $F$ (On input $< M,w >$ )
2:     Construct $M_1$.
3:     Construct $T_{<M,w>}$.
4:     Write $< M_1, T_{<M,w>} >$ on tape.
5:     Halt.
```

Proof. We proved that $\overline{A_{TM}} \leq_m EQ_{TM}$. Thus, by Theorem 6.2, $A_{TM} \leq_m \overline{EQ_{TM}}$. But then, if $\overline{EQ_{TM}}$ were decidable, so will be A_{TM}. So, $\overline{EQ_{TM}}$ is undecidable. □

Theorem 6.9. *Let*

$$REG_{TM} = \{< M >: M \text{ is a Turing machine and } L(M) \text{ is regular}\}.$$

Then, REG_{TM} is undecidable.

Proof. We demonstrate the statement by showing that A_{TM} is Turing-reducible to REG_{TM}.

Define the function $f : \{0,1\}^* \to \{0,1\}^*$, $f(< M,w >= N_{M,w}$, where $N_{M.w}$ is the Turing machine described in Algorithm 24. Then, if M accepts w, $L(N_{M.w}) = \{0,1\}^*$, a regular language. If M does not accept w, $L(N_{M,w}) = \{0^n1^n : n \in \mathbb{N}\}$, which is not regular. Therefore,

$$< M,w > \in A_{TM} \longleftrightarrow f(< M,w >) \in REG_{TM}.$$

Function f is computed by a Turing machine that on input $< M,w >$, writes a description of $< N_{M,w} >$ on tape, and halts.

Therefore, $A_{TM} \leq_m REG_{TM}$. But since A_{TM} is undecidable, REG_{TM} is also undecidable.

□

Corollary 6.3. *$\overline{REG_{TM}}$ is not Turing-recognizable.*

6.4 Rice Theorem

Section 6.3 shows quite a few undecidable problems. However, the list of undecidable problems is much larger. This section discusses the Rice Theorem, a central result that shows the incredible pervasiveness of undecidability.

Algorithm 24 Turing machine $N_{M,w}$ in proof of Theorem 6.9

```
procedure $N_{M,w}$ (On input $x$ )
    if $x = 0^n 1^n$ then
        Accept.
    else
        Run $M$ on $w$.
        if $M$ accepts then
            Accept.
```

Definition 6.4.1. By a *property* of a language we mean a predicate

$$P : \mathscr{P}(\Sigma^*) \to \{T, F\},$$

where Σ is an alphabet.

Example 6.4.1. Examples of properties P defined on $\mathscr{P}(\{0,1\}^*)$ are

1. $P(L) =:$ "L contains the string 0101;"
2. $P(L) =:$ "L contains all even palindromes, "
3. $P(L) =:$ "L does not contain strings that start and end with 0,"
4. $P(L) =:$ "L does not contain strings that start or end with 0."

We write $P(L)$ if L satisfies predicate P, and $\neg P(L)$, if it does not.

We concentrate on properties of the languages of Turing-machines. This is, properties of Turing-recognizable (or recursively enumerable) languages.

Definition 6.4.2. A property P satisfied by a Turing-recognizable language is said to be *non-trivial* if there are Turing-recognizable languages L_1 and L_2 such that $P(L_1)$ and $\neg P(L_2)$.

Example 6.4.2. Given a property P and a nonempty Turing-recognizable language L_1 that satisfies it, proving that the property is non-trivial amounts to proving that $L_2 = \overline{L_1}$ is nonempty and Turing-recognizable; or what is the same,

$$L_1 \neq \emptyset \wedge L_1 \neq \Sigma^* \wedge L_1 \text{ co} - \text{Turing recognizable.}$$

All predicates in Example 6.4.1 are non-trivial properties on the class of all Turing-recognizable languages. For instance, if $P(L) =:$ "L contains the string 0101," let L_1 be the language encoded by the regular expression $(0 \cup 1)^* 0101 (0 \cup 1)^*$. Since $0101 \in L_1$, $L_1 \neq \emptyset$ and since $1111 \notin L_1$, $L_1 \neq \Sigma^*$. And since L_1 is regular, is Turing-decidable and so, co-Turing recognizable.

If, on the other hand, $L_1 = \emptyset$, then L_1 is recognized by the Turing machine that rejects all inputs, and $L_2 = \overline{L_1}$ is recognized by the Turing machine that accepts all its inputs. Thus, any property over $L = \emptyset$ is a non-trivial property.

Algorithm 25 Turing machine T_{M,w,M_1} in proof of Theorem 6.10

```
1: procedure $T_{M,w,M_1}$ (On input $x$ )
2:     Run $M$ on $w$.
3:     if $M$ accepts then
4:         Run $M_1$ on $x$
5:         if $M_1$ accepts then
6:             Accept.
```

Given a non-trivial property P over Turing-recognizable languages, and a Turing-recognizer M; we pose the question: "Does the language of M satisfies P?" Rice Theorem states that this question cannot be answered algorithmically.

Theorem 6.10 (Rice Theorem)**.** *Let P be a non-trivial property over the class of Turing-recognizable languages. Then, the language*

$$L_P = \{< M >: \ M \,\text{Turing} - \text{recognizer and}\, P(L(M))\},$$

is undecidable.

Proof. By Turing-reduction of A_{TM} to L_P. We consider two cases, namely $\neg P(\emptyset)$ is true and $P(\emptyset)$ is true.
Assume first that $\neg P(\emptyset)$ is true. Since P is non-trivial, there is a Turing machine $M_1 \in L_P$. We define the reduction

$$\begin{aligned} &f : A_{TM} \to L_P \\ &f(< M, w >) =< T_{M,w,M_1} > \end{aligned}$$

where T_{M,w,M_1} is the Turing machine defined in Algorithm 25.

Now, if $w \in L(M)$, then $L(M_1) = L(T_{M,w,M_1})$. This means that

$$< M, w > \in A_{TM} \to < T_{M,w,M_1} > \in L_P.$$

Conversely, if $w \notin L(M)$, then $L(T_{M,w,M_1}) = \emptyset$. Because of our assumption, it follows that

$$< M, w > \notin A_{TM} \to < T_{M,w,M_1} > \notin L_P,$$

This proves that f is a reduction. In order to show that f is a Turing-reduction, we construct a Turing machine that on input $< M, w >$, constructs $\neg P(\emptyset)$ true writes it on its tape, and halts. This construction is doable because M_1 is a fixed Turing machine in L_P.

The proof of the case $P(\emptyset)$ true, follows after a minor modification of T_{M,w,M_1}, which is left as an exercise. □

Corollary 6.4. *The next formal languages are undecidable,*

$$\begin{aligned}
&\{< M >: M \text{ Turing} - \text{recognizer and } L(M) \text{ contains the string } 0101\},\\
&\{< M >: M \text{ Turing} - \text{recognizer and } L(M) \text{ contains all even palindromes}\},\\
&\{< M >: M \text{ Turing} - \text{recognizer and } L(M) \text{ does not contain strings that}\\
&\qquad\qquad \text{start or end with } 0\},\\
&\{< M >: M \text{ Turing} - \text{recognizer and } L(M) \text{ does not contain strings that}\\
&\qquad\qquad \text{start and end with } 0\}.
\end{aligned}$$

Chapter 7

Time Complexity

We assume that the reader is familiar with the notion of time complexity of algorithms, including *big O* concept, algebra and notation. In this chapter we use these concepts to estimate the rate of growth in the number of steps of a Turing machine. Certainly, this requires that the Turing machine under scrutiny be a decider, otherwise, the number of steps could be infinite. The ultimate aim of time complexity theory is providing a lower bound for the complexity (*i.e.* number of steps) of deciding a decidable language.

7.1 Some Basic Elements

In theory of algorithms complexity is a function of the number of operations performed by an algorithm on an input. By analogy, the complexity of a decider is the number of transitions that it executes on an input, before halting. These numbers vary depending on the *size of the input*. The size of the input is a parameter associated with the input that significantly influences the number of transitions. This parameter is often the length of the input string but sometimes may be something different, as the a number of iterations that are necessary to reach a solution within pre-established approximation bounds.

Thus, given a decider D, the number of transitions can be modeled with a *transition count* function

$$\begin{aligned}&f_D : \mathbb{N} \to \mathbb{N}\\&f_D(n) = \text{ number of transitions of } D \text{ on an input of size } n.\end{aligned}$$

As in theory of algorithms, we use a set of functions defined in $\mathbb{N}$ and with values in the nonnegative real numbers, $\mathbb{R}_{\geq 0}$, to estimate the rate of growth of f_D. Some common members in this set are

$$\begin{aligned}
&g(n) = c,\ c \in \mathbb{R}_{\geq 0} \text{ constant},\\
&g(n) = \log(n),\\
&g(n) = n\log(n),\\
&g(n) = n^k,\ k \in \mathbb{N}, \text{ fixed but arbitrary},\\
&g(n) = b^n,\ b \in \mathbb{R}_{\geq 0}, \text{ fixed but arbitrary}.
\end{aligned}$$

Definition 7.1.1. A transition count function f_D is said to be $O(g)$ if and only if there is $n_0 \in \mathbb{N}$, and $a > 0,\ a \in \mathbb{R}$, such that

$$(\forall n \in \mathbb{N}),\ n \geq n_0 \longrightarrow f_D(n) \leq g(n).$$

By the time complexity of D it is understood $O(h)$, where

$$h = \min\{g :\ f_D \in O(g)\}.$$

In turn, the complexity of the language L is

$$\min\{O(g) :\ O(g) \text{ complexity of } D, \text{ where } D \text{ is a decider of } L\}.$$

Thus, as expected, the complexity of a problem varies with the discovery of more efficient algorithms.

7.2 Time Complexity of $\{0^n1^n :\ n \in \mathbb{N}\}$

As a way of illustrating the distinction between algorithm and problem complexity, we take a look into the time complexity of the decidable language

$$\{w \in \{0,1\}^* :\ w = 0^n1^n,\ n \in \mathbb{N}\}.$$

In order to do that we consider the deciders described in Algorithms 26 and 27; and estimate their rate of growth. We use the length of the input string, this is $n = |w|$, as the size of the problem. Let's first count the number of transitions of machine M_1.

1. Line 2, and the subsequent lines 3 and 4, if applied; take $O(n)$ transitions.
2. The while instruction in line 5 is repeated $O(n/2)$ times. At each repetition, M_1 executes the $O(n)$ transitions described in line 6. Thus, lines 5 and 6 take $O(n^2)$ transitions.
3. Line 7 takes $O(n)$ transitions.

Thus, decider M_1 halts in $O(n^2)$ transition steps. Let's count now the number of transitions of machine M_2.

1. Line 2, and the subsequent lines 3 and 4, if applied; take $O(n)$ transitions.
2. The while instruction in line 5 is repeated $O\log(n)$ times. At each repetition, M_1 executes the $O(n)$ transitions described in line 6. Thus, lines 5 and 6 take $O(n\log(n))$ transitions.
3. Line 7 takes $O(n)$ transitions.

So, decider M_2 halts in $O(n\log(n))$ transition steps. Since the minimum of $O(n^2)$ and $O(n\log(n))$ is $O(n\log(n))$, that is the complexity of problem $\{0^n1^n : n \in \mathbb{N}\}$. This complexity estimate will change if a decider with a lower order of transition step is found.

Algorithm 26 First Decider

```
1: procedure $M_1$(On input w )
2:     Scan across the tape.
3:     if A 0 is found to the right of a 1 then
4:         Reject.
5:     while The tape contains at least a 0 and a 1 do
6:         Scan across the tape, crossing off a single 0 and a single 1.
7:     if The tape has a 0 or a 1 then
8:         Reject.
9:     Accept.
```

Algorithm 27 Second Decider

```
1: procedure $M_2$(On input w )
2:     Scan across the tape.
3:     if A 0 is found to the right of a 1 then
4:         Reject.
5:     while The tape contains at least a 0 and a 1 do
6:         if The length of the string of 0's and 1's is odd then
7:             Reject.
8:         Scan across the tape crossing off every other 0 and every other 1.
9:     if The tape has a 0 or a 1 then
10:        Reject.
11:    Accept.
```

7.3 Complexity of Multi-tape Deciders

As in the case of a single-tape decider, the time complexity of a multi-tape decider is the number of transition step that the machine takes to halt on a given input. The only difference is that multi-tape deciders operate simultaneously on more than one tape. The next theorem compares the time complexity of a multi-tape decider with that of its equivalent single-tape decider.

Theorem 7.1. *Let M be a k-tape decider that halts in $O(g(n))$ transition steps, where $g(n) \geq n$. Then, there is an equivalent single-tape decider D that halts in $O(g(n)^2)$ transition steps.*

Proof. The equivalent single-tape decider D, simulates the k tapes of M by splitting its single tape into k segments. At each step D scans across its tape and updates one of the segments. This means that in order to simulate M, D uses a tape of length $O(kg(n))$. Since D scans this tape $O(g(n))$ times, it halts in

$$O(kg(n)g(n)) = O(g(n)^2)$$

transition steps. □

Remark 7.3.1. While each k-tape decider can be simulated with a single-tape decider, the opposite is not true. This is, some single-tape Turing-machines do not have an equivalent multi-tape Turing machine. Thus, multi-tape complexity measures are restricted to a subset of decidable problems.

Algorithm 28 2-tape Decider

```
1: procedure M_3(On input w )
2:     Scan across the tape.
3:     if  A 0 is found to the right of a 1 then
4:         Reject.
5:     Scan across Tape 1 copying each 0 on Tape 2, until the first 1 is found.
6:     Scan the rest of Tape 1 crossing off a 0 in Tape 2 for each 1 read.
7:     if All 0's are crossed off and there is a 1 left then
8:         Reject.
9:     else if All 1's are read but a 0 in Tape 2 is not cross off  then
10:        Reject.
11:    Accept.
```

Example 7.3.1. Algorithm 28 is a 2-tape decider of $\{0^n1^n : n \in \mathbb{N}\}$. Its transition step count is as follows,

1. Line 2, and the subsequent lines 3 and 4, if applied; take $O(n)$ transitions.
2. The scan instructions of lines 5 and 6 take $O(n)$ transition steps.
3. Verifications in line 7 and 9 take $O(1)$ transitions each.

Thus, M_3 halts in $O(n)$ transition steps.

Remark 7.3.2. Notice that, as expected, the analysis in Example 7.3.1 is consistent with the statement of Theorem 7.1. Since M_3 is a 2-tape machine, Example 7.3.1 shows that the multi-tape complexity of problem $\{0^n1^n : n \in \mathbb{N}\}$ is $O(n)$.

7.4 Complexity of Nondeterministic Deciders

Unlike multi-tape Turing machines, the transition step count for nondeterministic machines requires a separate definition.

Definition 7.4.1. The transition count mapping of a nondeterministic decider N, is the mapping

$$f_N : \mathbb{N} \to \mathbb{N}$$
$$f_N(n) = \max\{\lambda : \lambda \text{ length of a computation branch of } N\}.$$

Remark 7.4.1. Notice that all computation branches of a nondeterministic Turing decider halt. Clearly, Definition 7.4.1 attempts to capture the worst case transition step count in the process of deciding whether a string does or doesn't belong to a language. If the string is rejected, each nondeterministic choice has to be run, and thus, the mapping in Definition 7.4.1 is the largest count of all possible choices. If the string is accepted, the worst case occurs when there is a single accepting branch and all rejecting nondeterministic choices are run before the accepting branch.

Theorem 7.2. *Let $O(g(n))$, $g(n) \geq n$, be the time complexity of a nondeterministic Turing machine. Then, its equivalent single-tape deterministic Turing machine halts in $O(2^{O(g(n)^2)})$ transition steps.*

Proof. By hypothesis, each computation branch of the nondeterministic decider N takes $O(g(n))$ transition steps. Let b be the maximum number of children that a node in the a node in this computation tree has. Then, the computation tree has at most $b^{O(g(n))}$ leaves. The deterministic single-tape D that is equivalent to N, simulates N by computing each node of each level of the computation tree, level by level. There are $O(b^{O(g(n))})$ nodes in the tree. Since D repeats each computation branch from the root to the node until reaching a halt state; reaching a halt state in a branch takes

$$1+2+\cdots+g(n) = \sum_{i=1}^{g(n)} i = \frac{g(n)(g(n)+1)}{2} = O(g(n)^2).$$

Therefore, D decides an input string of size n in

$$O(g(n)^2)O(b^{O(g(n))}) = O(g(n)^2 b^{O(g(n))})$$

transition steps. Let $z = g(n)^2 b^{O(g(n))}$, then

$$\log_2(z) = g(n) + \beta O(g(n)) = g(n) + O(g(n)),$$

where β is the constant of the change of base of logarithms. Thus,

$$z = 2^{(g(n)+O(g(n))} = 2^{g(n)O(g(n))},$$

which proves the statement. □

Remark 7.4.2. If the single-tape equivalent decider D keeps the configuration of each node of a level in the computation tree and use it to compute each configuration in the next level, then the time complexity of D is $O(2^{O(g(n))})$. In either case, the complexity of the single-tape decider is an exponential function of that of the nondeterministic decider.

7.5 The Class *P*

If the predicate that defines membership in a language admits a well-rounded logical or mathematical way of deciding its truth value, it is expected that the decider of that language will halt in *reasonable* time. By reasonable time it is loosely understood *polynomial* time, meaning $O(n^k)$ transition steps, k positive integer. In practice, if $k > 3$ most problems beyond a limit in size will not be decided in what everyone would agree is a reasonable time. Nonetheless, it is agreed that the limits in problem size that a polynomial time decider solves in reasonable time, are significantly higher that those of problems solved by exponential time deciders. For this reason, there is a particular interest in finding polynomial time deciders for languages.

Definition 7.5.1. P is the class of the languages that have a deterministic $O(n^k)$-time decider.

Theorem 7.3. *Every context-free language is in P.*

Proof. Let L be a context-free language. Then, there is a context-free grammar $G = (V,\Sigma,R,S)$ that generates L. We assume without loss of generality, that G is in Chomsky normal form. With this grammar, we construct the decider D described in Algorithm 29.

□

This method constructs bottom up, the inner nodes of the parse tree of the input string w, saving the variables in a dynamic programming table (T). The pseudocode segment that goes from lines 9 to 14 dominates the transition step growth of D. This segment consists of four nested loops but only three of them vary with n, as the innermost loop (line 12 to 14) is bounded above by $|R|$. Thus, D decides L in $O(n^3)$ time. Therefore, $L \in P$.

Example 7.5.1. As a way of illustrating the construction of the parse tree through Algorithm 29 consider

$$G = (\{S,A,B,C,D\},\{0,1\},R,S)$$
$$R = \{S \to AC|BC|\varepsilon, D \to AC|BC, A \to BD, B \to 0, C \to 1\}.$$

G is a context-free grammar in Chomsky normal form with $L(G) = \{0^n1^n : n \in \mathbb{N}\}$. Let $w = 0011$. Then, after executing steps 5 to 8 we get

Algorithm 29 Polynomial time decider of a CFL

```
 1: procedure D(On input w )
 2:     if $w = \varepsilon$ and $S \to \varepsilon \in R$ then
 3:         Accept.
 4:     else
 5:         for $i = 1$ to $n$ do
 6:             for each $A \in V$ do
 7:                 if $A \to w_i$ then $\in R$
 8:                     $T(i,i) \leftarrow A$.
 9:         for $l = 2$ to $n$ do
10:             for $i = 1$ to $n - l + 1$ do
11:                 $j \leftarrow i + l - 1$.
12:                 for each $A \to BC \in R$ do
13:                     if $T(i,k) = B \wedge T(k+1,j) = C$ then
14:                         $T(i,j) \leftarrow A$
15:     if $T(1,n) = S$ then
16:         Accept.
17:     else
18:         Reject.
```

B			
	B		
		C	
			C

.

After executing 9 to 15 with $l = 2$, we get,

B			
	B	S,D	
		C	
			C

.

After executing 9 to 15 with $l = 3$,

B		A	
	B	S,D	
		C	
			C

.

And finally, after executing 9 to 15 with $l = 4$,

Algorithm 30 Polynomial time decider of *PATH*

```
1: procedure $D$(On input $< G,s,t >$ )
2:     Place a mark on node $s$.
3:     for Each marked node $a$ do
4:         Scan all nodes $b$, $(a,b) \in E$
5:         if A node $b$ is unmarked then
6:             Mark node $b$
7:     if $t$ is marked then
8:         Accept.
9:     else
10:        Reject.
```

B		A	S,D
	B	S,D	
		C	
			C

.

Thus, 0011 is accepted.

Definition 7.5.2. Let G be a directed graph (*digraph*) and s and t nodes in G. We define

$$PATH = \{< G,s,t >: \text{ there is a directed path between } s \text{ and } t\}.$$

Theorem 7.4. *$PATH \in P$.*

Proof. Let $G = (V,E)$ and D be the decider described in Algorithm 30. Let $m = |V|$ be the size of the input. Then, instructions 3 and 4 are executed at most m times each. Thus, D halts in $O(m^2)$ transition steps. □

7.6 The Class *NP*

If the determination of the truth value of the predicate that defines membership in a formal language does not admit a well-rounded mathematical formulation, the decision on the membership of a string in a formal language is to be taken non-deterministically. For this, the decider searches over the space of variables of the predicate for a decision. Each such exploration is a branch in the computation tree of the nondeterministic machine. When implemented, nondeterministic machines are simulated with its single-tape deterministic equivalent. Consequently, according to Theorem 7.4.1, the actual time complexity of deciding this kind of problems is an exponential function of the nondeterministic time.

Definition 7.6.1. A formal language L is said to be nondeterministic polynomial (*NP*) if it can be decided by a nondeterministic polynomial time machine.

Algorithm 31 Verifier of L

```
1: procedure V(On input $< x, c >$ )
2:     if c divides x then
3:         Accept.
4:     else
5:         Reject.
```

Definition 7.6.2. A verifier of a decidable language L is a Turing machine V where

$$(\forall w \in \Sigma^*)\, w \in L \longleftrightarrow (\exists c \in \Sigma^*)\, V \text{accepts} < w, c > .$$

If this is the case, we say that L is verifiable by V. String c is called certificate.

Example 7.6.1. Let $L = \{x \in \mathbb{N} :\ x \text{ is a composite number}\}$. Then, algorithm 31 is a verifier for L.

Lemma 7.1. *Each $L \in NP$ is verifiable in polynomial time.*

Proof. Let $L \in NP$ and M be a polynomial time nondeterministic decider of L. Let V be a Turing machine simulation of a computation path generated by M, and c a string encoding a nondeterministic choice that ends in the accepting state. Then, for each $w \in \Sigma^*$, $w \in L$ if and only if there is a choice c of an accepting computing branch. Furthermore, since M computes in nondeterministic polynomial time, V halts in polynomial time. □

Definition 7.6.3. A binary formula ϕ is a well-formed logical expression of binary variables, and the operations of conjunction, disjunction and negation. A binary formula is said to be *satisfiable* if and only if there is an assignment of v of values in $\{0,1\}$ that make the formula true. We denote the assignment by $\phi(v)$ and define

$$SAT = \{\phi :\ (\exists v)\phi(v)\}.$$

Algorithm 32 Polynomial time decider of SAT

```
1: procedure S(On input $< \phi >$, $\phi$ a binary formula )
2:     Compute $n \leftarrow$ number of variables in $b$.
3:     Nondeterministically select an array $v$ of $n$ values in $\{0,1\}$
4:     if $\phi(v)$ is true then
5:         Accept.
6:     else
7:         Reject.
```

Algorithm 33 Verifier of *SAT*

```
1: procedure V(On input < $\phi, c$ >; $\phi$ n-variable Boolean formula, c array of n
   values in $\{0,1\}$ )
2:     if $\phi(c)$ is true then
3:         Accept.
4:     else
5:         Reject.
```

Algorithm 34 Decider for Proof of Theorem 7.6

```
1: procedure $T_{F,D_K}$(On input w )
2:     Run F on w.
3:     Retrieve $f(w)$ from the tape of F.
4:     Run $D_K$ onf $f(w)$.
5:     if $D_K$ accepts then
6:         Accept.
7:     else
8:         Reject.
```

Theorem 7.5. *$SAT \in NP$*

Proof. The Turing machine S described in Algorithm 32 is a nondeterminstic polynomial time decider of SAT. Algorithm 33 is a $O(n)$-time verifier of SAT. □

Lemma 7.2. *$P \subset NP$.*

Proof. It follows from the fact that deterministic Turing machines are a special case of nondeterministic Turing machines. □

Remark 7.6.1. A major unanswered question in complexity theory is whether $P \neq NP$. There is overwhelming evidence that this is indeed the case, but complexity theory has not reached yet a level of development enabling a sound theoretical answer to this question.

7.7 Cook-Levin Theory

Cook-Levin theorem is the most significant advance towards an answer of the $P \neq NP$ question.

Definition 7.7.1 (polynomial time reduction)**.** A language L is said to be reducible to a language K in polynomial time, denoted $L \leq_p K$, if there is a reduction $f : L \to K$ that can be computed in polynomial time.

Theorem 7.6. *If $L \leq_p K$ and $K \in P$, then $L \in P$.*

Proof. Let D_K be a polynomial time decider for K, and f a polynomial time reduction of L to K. Since f is a polynomial time reduction, is computed in polynomial time. Then, since for each $w \in L$, $f(w) \in K$. Decider T_{F,D_K} described in Algorithm 34 runs in polynomial time since D_K and F run in polynomial time. □

Definition 7.7.2. A language L is said to be NP-complete if

1. $L \in NP$
2. $\forall K \in NP,\ K \leq_p L$

Theorem 7.7. *If there is an L that is NP-complete and is also in P, then* $P = NP$.

Proof. If L is NP-complete, then each $K \in NP$, $K \leq_p L$. Thus, if L is also in P, K is in P because of Theorem 7.6. □

Theorem 7.8 (Cook-Levin Theorem)**.** *SAT is NP-complete.*

Proof. Theorem 7.5 demonstrates that $SAT \in NP$. It remains to prove that

$$\forall K \in NP,\ K \leq_p SAT.$$

This is, we have to construct a polynomial time reduction

$$\begin{aligned} &f : \Sigma^* \longrightarrow \Sigma^* \\ &f(w) = <\phi_w>, \end{aligned}$$

where ϕ_w is a satisfiable Boolean formula.

Since a language in NP can be decided in $O(n^k)$ time by a nondeterministic Turing machine, each branch of a computation on an input w can be represented in a $O(n^k) \times O(n^k)$ array $A_w = [A_w(i,j)]$, referred as *tableau*; where the path of configurations in a branch is written row-by-row. Since the maximum length of the configurations is bounded above, the symbol # is used to denote the first and last columns of A_w.Thus,

$$A_w(i,j) = \begin{cases} \text{a state symbol } q \in Q, \\ \text{a symbol } w \in \Gamma, \\ \text{the first or last column symbol \#.} \end{cases}$$

We define the Boolean variables $x_{i,j,s}$, $s \in Q \cup \Gamma \cup \{\#\}$; by the proposition

$$x_{i,j,s} =: \text{“}s = A_w(i,j).\text{”}$$

The Boolean formula ϕ_w is constructed with these variables, and as the conjunction

$$\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}.$$

Each of these formulas is defined next.

1. Definition of ϕ_{cell}. This formula is true only if A_w is well-defined.

$$\phi_{cell} =: \bigwedge_{1\leq i,j\leq c\cdot n^k}\left[\left(\bigvee_{s\in Q\cup\Gamma\cup\{\#\}} x_{i,j,s}\right)\wedge\left(\bigwedge_{s,t\in Q\cup\Gamma\cup\{\#\},s\neq t}\overline{x_{i,j,s}\wedge x_{i,j,t}}\right)\right].$$

2. Definition of ϕ_{start}. This formula is true only if the first row in A_w is the start configuration of the computation of K on input $w = w_1\cdots w_n$.

$$\phi_{start} =: x_{1,1,\#}\wedge x_{1,2,q_0}\wedge\cdots\wedge x_{1,j,w_{j-2}}\wedge\cdots\wedge x_{1,n+2,w_n}\wedge\sqcup\cdots\sqcup\wedge x_{1,c\cdot n^k,\#}.$$

3. Definition of ϕ_{move}. This formula is true only if each yield in the sequence of configurations corresponds to a Turing machine transition. Since each transition overwrites one symbol in the configuration and changes and moves the state either one place to the left of to the right, the correction of a yield can be verified by checking each 2×6 subarray in A_w. The 2×6 subarray whose leftmost upper element is $A_w(i,j)$ is denoted $W_{i,j}$ and referred as i,j-*window*. We defined the Boolean variable

$$U_{i,j} =: \text{“}W_{i,j}\text{ is consistent with a Turing machine transition,}'\text{”}$$

and set

$$\phi_{move} =: \bigwedge_{1<i,j\leq c\cdot n^k} U_{i,j}.$$

4. Definition of ϕ_{accept}. This formula returns true only if the last row in A_w is an accepting configuration. The formula is,

$$\phi_{accept} =: \bigvee_{1\leq i,j\leq c\cdot n^k} x_{i,j,q_a}.$$

Function f is a reduction because if $w\in K$, then there is an accepting computing branch whose representation as a tableau A_w, renders ϕ_w true. Thus, f maps K to SAT. Conversely, if $w\notin K$, all branches in the computation of w are reject w, and each tableau A_w produces a false ϕ_{accept}. Thus, there is no satisfying assignment for the variables of ϕ_w, or what is the same, $\phi_w\notin SAT$. Therefore, f maps $\overline{K}$ to $\overline{SAT}$.

Algorithm 35 Computation of function $w\to\phi_w$ in Theorem 7.8

```
1: procedure F(On input w )
2:     $V \leftarrow$ accepting branch of $T_K$.
3:     Run $V$ on input $w$, constructing $A_w$.
4:     Construct $\phi_w$.
5:     Write $\phi_w$ on tape.
6:     Halt.
```

Finally, in order to see that f is computable in polynomial time, we consider the Turing machine F described in Algorithm 35. Step 3 and 4 dominate, and both take $O(n^{2k})$ time to complete. Thus, f is a polynomial time reduction. $\square$