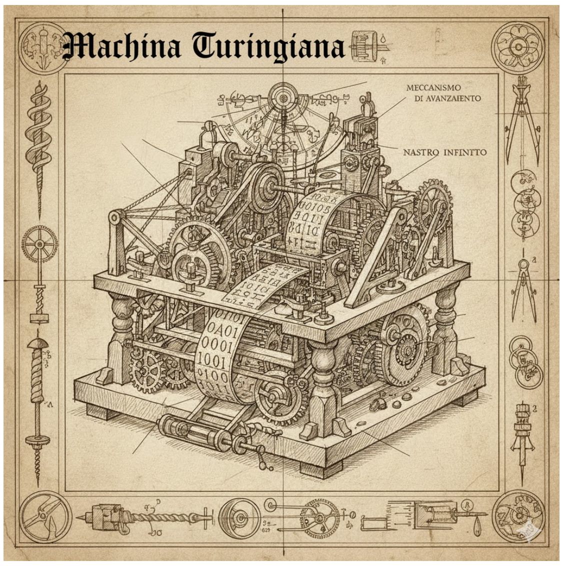
Formal Languages and Calculability

Lecturer: Prof. Dr. Luca Doria
Johannes Gutenberg University Mainz

Summer Term 2026

These course notes summarize a fundamental topic in Computer Science: computability. The aim of this theory is to understand what a computer can and cannot do — though its scope is much broader. In particular, computability explores the limits of mechanical calculation, which is not only what computers do, but also a part of human reasoning. This has profound implications beyond computer science, reaching into mathematics, philosophy, linguistics, and physics.



Contents

1 Introduction

1.1 Computability

Computability Theory, (called also Recursion Theory by mathematicians), is a foundational branch of mathematical logic and theoretical computer science that investigates the fundamental question: **What does it mean for a function to be computable?** It tries to determine which problems can be solved by a mechanical or algorithmic process (like a real computer would do) and which cannot. The field was pioneered in the 1930s by mathematicians like Alan Turing (1912-1954), Alonzo Church (1903-1995), Kurt Gödel (1906-1978), and Stephen Kleene (1909-1994), well before the advent of modern electronic computers.

Computability Theory is the theoretical backbone of computer science. Its primary contribution is establishing the limits of computation. This is an endeavour which extends way beyond computers, involving physical systems and their governing laws in general. Computability also defines algorithms: It provides a precise, formal model—most famously the Turing Machine—to define what an algorithm is.

Another central topic of this theory are Decidability and Undecidability: The theory differentiates between problems that are decidable (solvable by an algorithm that always "halts") and those that are undecidable (problems for which no algorithm can ever give a correct "yes" or "no" answer). The classic example of an undecidable problem is the Halting Problem, which asks if it's possible to determine, given a program and its input, whether the program will eventually stop or run forever. This demonstrates that there are inherent, permanent limitations to what computers can achieve.

Beyond mathematics and theoretical computer science, computability theory raises profound questions across disciplines:

- Philosophy of mind (The Computational Theory of Mind): The Turing Machine model provides a powerful framework for discussing the nature of intelligence. The Computational Theory of Mind suggests that the human mind itself is a kind of computer. The Church-Turing Thesis, therefore, becomes central to debates about Artificial Intelligence and whether a machine can achieve “real” consciousness or solve problems inherently beyond standard algorithms.

- Philosophy of science: Computability relates to the fundamental nature of the universe. The concept of a "computable universe" asks whether the laws of physics and the evolution of the cosmos can be simulated or predicted by a universal (quantum?) computer.

- The Nature of Mathematics: By identifying truly non-computable problems, the theory defines the boundaries of algorithmic knowledge, impacting the field of epistemology by formalizing what we can know about the output of a mathematical system purely through mechanical process.

1.2 Models of Computation and Languages

Theoretical computer science links models of computation (formal, mathematically defined abstract machines) to formal languages (sets of strings recognized by those machines). This relationship is precisely characterized by the **Chomsky Hierarchy**, which organizes these models and the languages they can process based on their computational power. This might sound quite abstract, but the core idea is simple: consider a sentence written in a specific language. The fundamental question then becomes, which kind of "computer" would be able to understand it?

For languages with a simple, rigid structure, even a very basic, rudimentary machine might suffice. However, as the complexity of the language increases, we will necessarily require a much more sophisticated computational device.

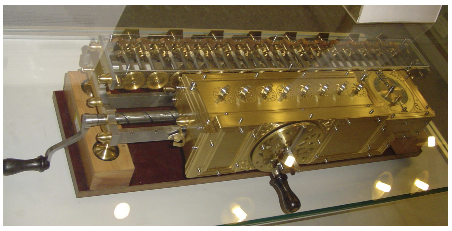
Studying different types of machines, or models of computation, is essential for understanding their inherent limitations and identifying the specific characteristics that make one machine more powerful than another. What is it that makes the computers we use every day so special? Could we build machines that are even "better" and calculate more if we based them on a different computational model? Or does our current model represent the ultimate, fundamental limit of automated calculation and algorithm execution?

To answer these questions, we will begin with basic mathematical tools and a very simple model of computation, such as boolean circuits. After recognizing its limitations, we will sequentially move to more powerful models. As we proceed up the hierarchy of computational power, we will also formally define the languages these machines can process, ultimately drawing connections to our own natural language.

The central results of computability theory often yield binary answers: can a certain computational model solve a given problem (yes or no)? This means that computability theory does not address the more subtle question: "how difficult is this problem?" Attempting to answer that challenge moves us into the realm of **Computational Complexity Theory**, an extremely important topic that logically follows computability.

1.3 Brief History of Mechanical Computation

The need to automate calculation is as old as civilization itself. Early people relied on rudimentary tools like sticks and the abacus to manage commerce. An extraordinary example of complex early computation is represented by the Antikythera Mechanism (circa 2nd century BCE), a device discovered off the coast of Greece. Composed of dozens of interlocking bronze gears, it functioned as an astronomical calculator, capable of predicting solar and lunar eclipses and planetary movements with remarkable accuracy. This device demonstrates that the conceptual basis for mechanical calculation has existed for over two millennia!

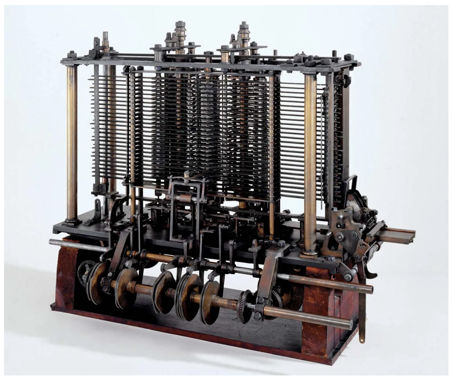

Reproduction of a Pascaline (Deutsches Museum)

The next major step occurred in the 17th century with the invention of automated calculating aids: John Napier's (1550-1617) logarithms and bones [1], Blaise Pascal's (1623-1662) mechanical adding machine (the Pascaline), and Gottfried Wilhelm Leibniz's (1646-1716) Stepped Reckoner, which could perform even multiplications. These devices paved the way for the 19th-century invention of Charles Babbage (1791-1871) who designed the Difference Engine to automate the calculation of polynomial functions and, more importantly, the Analytical Engine. Babbage's collaborator, Ada Lovelace (1815-1852), realized the machine's true potential, describing how it could process more than just numbers but also symbols and logical operations. A. Lovelace effectively became the world's first person designing an algorithm for a computer! However, it was not until the 1930s, driven by a crisis in the foundations of mathematics, that the theoretical limits of computation were formally defined. The work of Gödel, Turing, and others created the abstract models that paved the way for the electronic com-

[1] The term "Napier's bones" refers to a clever manual calculating device invented by John Napier. Napier's bones (or Napier's rods) were a set of numbered rods used as a manipulative tool to perform multiplication and division. The rods were typically made of wood, bone, or ivory (hence the name "bones") and had multiplication tables engraved on their faces. By placing the correct rods side-by-side, a user could read off the partial products and sum them up quickly, even for large numbers, thus mechanizing a key part of the calculation process. They were widely used by engineers and merchants in the 17th century.

puters we use today.

Following the foundational theoretical work, the development of practical computing accelerated—most notably through the pioneering efforts of Konrad Zuse. In 1941, Zuse completed the Z3, the first fully functional, programmable, electromechanical digital computer. Although his work took place during wartime and remained relatively isolated from British and American developments, the Z3 introduced concepts such as binary arithmetic and floating-point calculations that would become essential to later computers.


The Babbage analytical engine

In USA, ENIAC was built in 1946, one of the first large-scale electronic general-purpose computers. ENIAC was based on thousands of vacuum tubes and programming was equivalen to “rewiring” the machine. This limitation was addressed by the introduction of the stored-program concept (like a Turing machine, actually!), first implemented in 1948 in the United Kingdom. These machines marked a key shift: instructions could now be stored in the computer’s memory rather than manually encoded.

The UNIVAC I (1951) was the first computer designed for commercial business use, and it brought computing out of universities and military institutions for the first time. The indroduction of transistors in the mid-1950s made computers smaller, faster, more reliable, and energy-efficient. By the late 1950s and early 1960s, integrated circuits began to replace individual transistors, marking yet another leap in miniaturization and efficiency. The rest is history, ending with the machine your are likely using to read these lines.

2 Mathematical Tools

In this chapter we shortly summarize the basic mathematical tools, structures, and procedures we will need for discussing calculability.

2.1 Elementary Structures

2.1.1 Sets

A very basic object is the set:

Sets

A **set** is a group of objects represented as an unit.

Note that this definition might sound circular: if “a set is a group”, what is a “group”? We have to give up with precise basic definitions at some point and assume an “intuitive” understading. This is what we do with sets: we assume that everybody understands what we are talking about. Having these intuitive basic definitions, hopefully we can construct rigorously all the rest.

Sets can contain certain objects (called *elements* or *members*) and also other sets. We indicate a set with the notation

$$A = \{a_1, a_2, a_3, ...\} \quad ,$$

where A is the name of the set and a_i are the contained elements.
$a_1 \in A$ indicates membership: a_1 belongs to the set A.
If B is a subset of A, we write: $B \subseteq A$. The subset symbol includes the case where A=B (the sets are identical). If B is strictly included in A we say that B is a *proper subset* and

Subsets

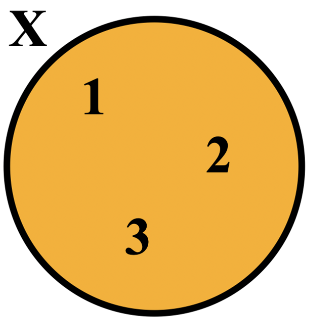
we write $A \subset B$.

Multisets

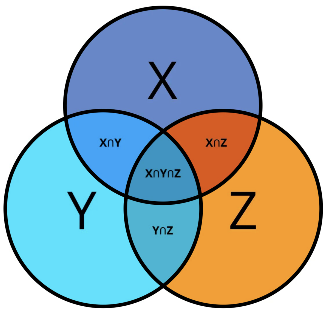
Usually, we distinguish sets from *multisets*. In sets, the repetition of a member does not define a different set, so for example $\{a_1, a_2\} = \{a_1, a_1, a_1, a_2\}$. In the case of multisets, the sets in the last example are different.

Venn-Diagrams

Another notation for indicating sets, is based on Venn diagrams (John Venn, 1834–1923):



2.1.2 Set Operations and other Properties



- **The empty set:** $\emptyset$
- **Intersection:** $A \cap B = \{x : x \in A \text{ and } x \in B\}$
- **Union:** $A \cup B = \{x : x \in A \text{ or } x \in B\}$
- **Complement:** $\bar{A} = \{x : x \notin A\}$. The complement is *involutive:* $\bar{\bar{A}} = A$
  Usually we think that a set is embedded in another "set of all the sets" which we can call the "universum" U, therefore $\bar{A} \cup A = U$, $\bar{U} = \emptyset$.
  The complement of subsets satisfies: $A \subset B$ implies $\bar{B} \subset \bar{A}$.

- **De Morgan's Laws:**
  $A \bar{\cap} B = \bar{A} \cup \bar{B}$
  $A \bar{\cup} B = \bar{A} \cap \bar{B}$

- **The power set:** Given a set A, we can define the "power set" operation which returns a set containing all the possible subsets of A, including A and $\emptyset$: $\mathcal{P}(A) = \{B : B \subset A\}$.

- **Cardinality:** it is the number of elements in a set. If a set A has cardinality |A| = n, the power set has cardinality 2^n, justifying the previous notation for the power set.

2.1.3 Sequences and Tuples

Sets were notationally defined with curly brackets $A = \{...\}$ and they were just collections of objects and the order of appearence of the members does not matter, *e.g.* $\{1,2\} = \{2,1\}$. If the order matters, we are talking about **sequences** which are identified with normal brackets: $S = (1,2,3,...)$. Sequences can be finite (also called **tuples**) or infinite. More in general we refer to k-tuples: for example, a 2-tuple is an ordered pair of objects, like $(1,2)$.
An important operation is the **cartesian product** between two sets A and B. The cartesian product produces a new set composed by all the 2-tuples which can be constructed picking one element from A and one from B. For example, if $A = \{a,b,c\}$ and $B = \{1,2\}$ we have

$$A \times B = \{(a,1),(a,2),(b,1),(b,2),(c,1),(c,2)\} \quad .$$

From three sets A,B,C we can also create 3-tuples calculating $A \times B \times C$ and so on.

2.1.4 Functions

Functions are central to mathematics and are eventually also the object of study of calculability (which functions are calculable? which ones are not?). A function is a procedure (a "calculation"? an "algorithm"?) which from an input x produces an output y. If we think of $x \in X$ and $y \in Y$ as elements of two different sets, a function f can be regarded as a relationship or a *mapping* among sets:

$$f : X \to Y \quad .$$

Commonly X is referred to as the *domain* and Y the *range* (or image, or codomain) of the function. A useful classification of functions is the following (see also Fig. 2.1):

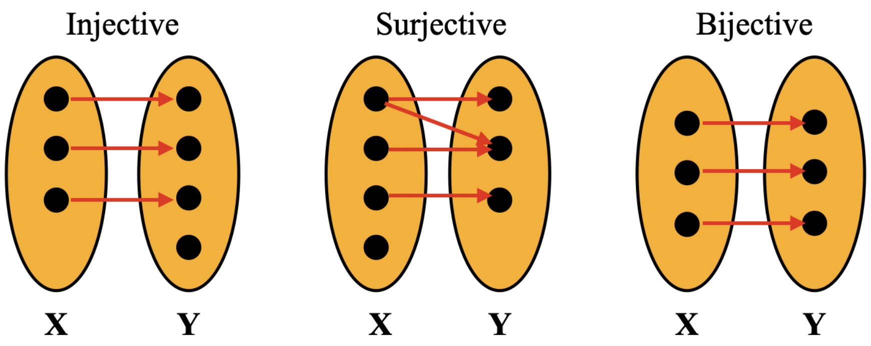


Figure 2.1: The three families of functions most relevant in theorems and applications. For an injective function, one element of the domain corresponds to only one element of the image. For surjective functions, this one-to-one correspondence is not required, but all the elements of the image must correspond to some element of the domain. Bijective functions combine the properties of injective and surjective functions: the whole image is covered and the correspondence is one-to-one.

$$\begin{aligned}
&\text{Injective:} \quad f : A \to B \text{ is injective} \iff \forall x_1, x_2 \in A,\ f(x_1) = f(x_2) \Rightarrow x_1 = x_2,\\
&\text{Surjective:} \quad f : A \to B \text{ is surjective} \iff \forall y \in B,\ \exists x \in A \text{ such that } f(x) = y,\\
&\text{Bijective:} \quad f : A \to B \text{ is bijective} \iff f \text{ is injective and surjective.}
\end{aligned}$$

2.1.5 Numbers

A common way to define natural (integer) numbers is through the Peano Axioms [1]. Let 0 denote the first natural number, S the *successor function*, and $\mathbb{N}$ the set of natural numbers. The Peano axioms are:

- **(A1:)** $0 \in \mathbb{N}$.
- **(A2:)** $\forall n \in \mathbb{N} \left(S(n) \in \mathbb{N}\right)$.
- **(A3:)** $\neg\exists n \in \mathbb{N} \left(S(n) = 0\right)$.
- **(A4:)** $\forall a, b \in \mathbb{N} \left(S(a) = S(b) \to a = b\right)$.
- **(A5:)** (Induction axiom) If $X \subseteq \mathbb{N}$ satisfies $0 \in X$ and $\forall n \in X\ S(n) \in X$, then $X = \mathbb{N}$.

[1] Giuseppe Peano (1858-1932)

The induction axiom was formulated in terms of set theory, but it can also be formulated as a scheme for every formula $\varphi(n)$ (possibly with parameters):

$$\big(\varphi(0) \land \forall n\ (\varphi(n) \to \varphi(S(n)))\big) \implies \forall n\ \varphi(n).$$

Number sets we will consider are:

- **Integer numbers** $\mathbb{N} = \{0, 1, 2, 3, ...\}$
- **Relative numbers** $\mathbb{Z} = \{0, -1, 1, -2, 2, ...\}$
- **Rational numbers** $\mathbb{Q} = \{a/b : a \in \mathbb{Z} \text{ and } b \in \mathbb{N}\}$
- **Real numbers** $\mathbb{R} = \mathbb{Q} \cup \mathbb{I}$ where $\mathbb{I}$ are the *irrational numbers*.

Note the somewhat surprising fact[2] that $|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| < |\mathbb{R}|$.
We can actually state the following theorem:

Theorem 2.1.1: Cantor (1874)

There is no bijection $f : \mathbb{N} \to (0, 1) \in \mathbb{R}$ and $|(0, 1)| > |\mathbb{N}|$

Proof. The theorem does not just say that $|\mathbb{R}| > |\mathbb{N}|$ but that even a subset of the real numbers already contains more numbers than $\mathbb{N}$. The proof follows the now classic "diagonal" method devised by Cantor later in 1891 (the first oroginal proof was analysis-style and based on countable sets if intervals on the real line).
Let's assume the numbers in $(0, 1)$ were countable: $r_1, r_2, ...$ We can represent the decimal numbers of the i-th number r_i with binary digits:

$$r_i = 0, b_{11} b_{i2} b_{i3} ... \quad ,$$

obtaining a sequence:

	1	2	3	...
r_1	r_{11}	r_{12}	r_{13}	...
r_2	r_{21}	r_{22}	r_{23}	...
...	...	...	...	...

[2] The uncountability of the real numbers ($|\mathbb{N}| < |\mathbb{R}|$) was proved in 1874 by G. Cantor (1854-1918).

We define now a new number $r = 0, c_1c_2c_3...$ with $c_i = 1 - b_{ii}$. This number differs from r_1 by the first 'bit' after the comma, from r_2 by the second bit and so on. This means that such a number cannot be part of the table, therefore we cannot "count" the real numbers: they must be 'more' than the integers. □

2.1.6 Graphs

A graph G is defined as a set containing other two sets: $G = \{V, E\}$. V is the set of *vertices* and E is the set of *edges*. Graphically we can picture vertices as dots and edges as segments connecting couples of dots. We distinguish between *directed* and *undirected* graphs. In directed graphs, the edge has also a "direction" while in undirected graphs this is not the case. We can represent an edge as a pair of nodes (the nodes connected by the edge). If this pair is ordered (it is a 2-tuple) for each edge in a graph, then the graph is directed. The nodes can carry additional information, like their name and other mathematical structures.

2.2 Logic

Logic is the branch of mathematics that studies the principles of valid reasoning. It provides a formal language to express statements precisely and a system of rules to determine whether conclusions follow from given assumptions. In logic, the basic objects are *statements* (or *propositions*): sentences that are either true or false (but not both!). There are different kind of logic and each can have some limitations. In the following we describe two of them which are commonly considered in mathematics and computer science.

2.2.1 Propositional and Boolean Logic

In this logic, the propositions can be combined using the following operations:

Negation:	$\neg P$	"not P"
Conjunction:	$P \wedge Q$	"P and Q"
Disjunction:	$P \vee Q$	"P or Q" (inclusive)
Exclusive or:	$P \oplus Q$	"either P or Q, but not both"
Implication:	$P \rightarrow Q$	"if P, then Q"
Biconditional:	$P \leftrightarrow Q$	"P if and only if Q"

Truth Table for Basic Logical Operators

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$
T	T	F	T	T	T
T	F	F	F	T	F
F	T	T	F	T	T
F	F	T	F	F	T

P	Q	$P \leftrightarrow Q$	$P \oplus Q$	P NAND Q	P NOR Q
T	T	T	F	F	F
T	F	F	T	T	F
F	T	F	T	T	F
F	F	T	F	T	T

An interesting property which is also relevant for computer science is **functional completeness**. A set of logical operations (or connectives) is called *functionally complete* if it can be used for constructing all the other operations. In particular, we can ask what is the *minimal* set of operations able to express all the others. It turns out that in propositional logic only *one* operator is needed! Among **two-input logical operators**, only `NAND` and `NOR` can generate all other logical operations by themselves:

NAND (Sheffer Stroke): $P \uparrow Q \;\equiv\; \neg(P \wedge Q)$

$$\begin{aligned}
\neg P &= P \uparrow P \\
P \wedge Q &= (P \uparrow Q) \uparrow (P \uparrow Q) \\
P \vee Q &= (P \uparrow P) \uparrow (Q \uparrow Q) \\
P \rightarrow Q &= P \uparrow (Q \uparrow Q)
\end{aligned}$$

NOR (Peirce Arrow): $P \downarrow Q \;\equiv\; \neg(P \vee Q)$

$$\begin{aligned}
\neg P &= P \downarrow P \\
P \vee Q &= (P \downarrow Q) \downarrow (P \downarrow Q) \\
P \wedge Q &= (P \downarrow P) \downarrow (Q \downarrow Q) \\
P \rightarrow Q &= P \downarrow (P \downarrow Q)
\end{aligned}$$

Larger sets can be found (like e.g. $\{\neg, \vee\}$, $\{\neg, \wedge\}$, $\{\neg, \rightarrow\}$) but no unary operators can act as functionally complete connective. In real microprocessors, logic gates are indeed often implemented starting from NAND and NOR gates.

Boolean logic is similar to propositional logic but usually involves only the $\neg$, $\wedge$, $\vee$ operators and not “inference rules”.

2.2.2 Predicate Logic

Predicate Logic is an extension of propositional logic that allows us to reason about objects, properties, and relationships. While propositional logic treats an entire statement as a single atomic unit, predicate logic breaks the statement down into a predicate (a property) and its arguments (the objects it applies to). This allows for much detailed analysis and formalization of complex mathematical and natural language statements.

Predicate logic uses the following kinds of symbols:

- **Variables:** $x, y, z, x_1, x_2, \ldots$
- **Constants:** $a, b, c, 0, 1, \ldots$
- **Function symbols:** $f, g, h, +, \times, \ldots$
- **Predicate (relation) symbols:** $P, Q, R, =, <, \ldots$
- **Logical connectives:** $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- **Quantifiers:** $\forall$ ("for all"), $\exists$ ("there exists")
- **Punctuation:** parentheses () and commas

3. Logical Equivalences and Laws:

- **Double Negation:** $\neg(\neg\varphi) \equiv \varphi$
- **Commutativity:** $\varphi \wedge \psi \equiv \psi \wedge \varphi,\ \varphi \vee \psi \equiv \psi \vee \varphi$
- **Associativity:** $(\varphi \wedge (\psi \wedge \theta)) \equiv ((\varphi \wedge \psi) \wedge \theta)$
- **Distributivity:** $\varphi \wedge (\psi \vee \theta) \equiv (\varphi \wedge \psi) \vee (\varphi \wedge \theta)$
- **De Morgan:** $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi, \quad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$
- **Implication:** $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- **Quantifier Negation:** $\neg(\forall x\, \varphi) \equiv \exists x\, (\neg\varphi), \quad \neg(\exists x\, \varphi) \equiv \forall x\, (\neg\varphi)$
- **Quantifier Distributivity (valid where x not free in ψ):** $\forall x(\varphi \wedge \psi) \equiv (\forall x \varphi) \wedge \psi$, $\exists x(\varphi \vee \psi) \equiv (\exists x \varphi) \vee \psi$

%begindescription[labelwidth=3cm]

2.3 Predicate Logic Examples

Basic Sentences with Quantifiers and Connectives

(1) $\forall x\, P(x)$ "Every x has property P."

(2) $\exists x\, P(x)$ "There exists at least one x such that $P(x)$ holds."

(3) $\neg\exists x\, P(x)$ "There is no x such that $P(x)$."

(4) $\forall x\, \neg P(x)$ "Every x fails to have property P."

(5) $\forall x\, (P(x) \to Q(x))$ "For all x, if $P(x)$ holds then $Q(x)$ holds."

Nested Quantifiers and Parentheses

(6) $\forall x\, \exists y\, (R(x, y))$ "For every x, there exists a y such that $R(x, y)$."

(7) $\exists y\, \forall x\, (R(x, y))$ "There exists a y that is related to all x."

(8) $\forall x\, \forall y\, (R(x, y) \leftrightarrow R(y, x))$ "The relation R is symmetric."

(9) $\forall x\, \forall y\, \forall z\, ((R(x, y) \wedge R(y, z)) \to R(x, z))$ "The relation R is transitive."

(10) $\exists x\, \forall y\, (P(y) \to R(x, y))$ "There exists an x such that for every y, if y has property P, then x is related to y."

Examples Involving Equality and Arithmetic-Like Symbols

(11) $\forall x\, \exists y\, (x < y)$ "Every number has a larger number."

(12) $\exists x\, \forall y\, (y \geq x)$ "There exists a smallest number."

(13) $\forall x\, \exists y\, (y = S(x))$ "Every element x has a successor $S(x)$."

(14) $\forall x\, (x + 0 = x)$ "Zero is the additive identity."

(15) $\forall x\, \forall y\, (x + y = y + x)$ "Addition is commutative."

Mixed Logical Operators

(16) $\forall x\,((P(x) \lor Q(x)) \to R(x))$ "If x has property P or Q, then x also has R."

(17) $(\exists x\, P(x)) \land (\exists x\, Q(x))$ "There exists some x with property P and some (possibly different) x with property Q."

(18) $\exists x\,(P(x) \land Q(x))$ "There exists at least one x that has both properties P and Q."

(19) $\forall x\,(\neg P(x) \lor Q(x))$ Equivalent to: $\forall x\,(P(x) \to Q(x))$.

(20) $\forall x\,\exists y\,((P(x) \land R(x,y)) \to Q(y))$ "For every x, there exists a y such that if x has property P and is related to y, then y has property Q."

Quantifier Negation Examples

(21) $\neg\forall x\, P(x) \equiv \exists x\, \neg P(x)$ "Not all x satisfy P" means "There exists some x that does not satisfy P."

(22) $\neg\exists x\, P(x) \equiv \forall x\, \neg P(x)$ "No x satisfies P" means "Every x fails to satisfy P."

More Complex Examples

Example: "Everyone who loves someone is loved by someone."

$$\forall x\,\Big[(\exists y\,(Loves(x,y))) \to (\exists z\,(Loves(z,x)))\Big]$$

Example: For every human x and every y , if y is the mother of x , then y loves x.

$$\forall x\,\forall y\,\Big((Human(x) \land Mother(y,x)) \to Loves(y,x)\Big)$$

2.4 Proofs

In mathematics, a proof is a logical argument that demonstrates the truth of a statement beyond any doubt. It begins from previously accepted facts—such as **axioms**, **definitions**, or earlier theorems—and proceeds through a sequence of justified steps, each following from the previous by *rules of logic*. The purpose of a proof is to establish certainty: a mathematical statement is considered true only if it can be proven within an accepted *logical framework*. Proofs distinguish mathematics from empirical sciences, where evidence can support but not guarantee truth.

A proof might even be very compact and short, but the real challenge is to find them! In the following we will look at an attempt to classify proofs according to certain techniques.

2.4.1 Proofs by Construction

This kind of proof method answer the quesiton “does this object exist?” directly constructing it. A very simple example is the following:

Theorem: there is a prime number which is even

Proof: starting from the smallest prime number we find right away that 2 is prime and even. Providing a concrete example proves the theorem.

2.4.2 Proofs by Contradiction

In this powerful technuque, we assume that the theorem we would like to prove true is actually false! Following this assumption, we check if we are lead to a contradiction. If we eventually find a contradiction, then the theorem must have been true. A classic example is the proof of the irrationality of $\sqrt{2}$ which according to the lore, rattled the Pithagorean school in the 6th century BCE:

Theorem 2.4.1:

The number $\sqrt{2}$ is irrational.

Proof. We prove the claim by contradiction. Assume, for the sake of contradiction, that $\sqrt{2}$ is rational. Then there exist integers p and q with $q \neq 0$ such that

$$\sqrt{2} = \frac{p}{q}.$$

We may moreover assume that the fraction $\frac{p}{q}$ is in lowest terms, i.e. $\gcd(p, q) = 1$.

Squaring both sides gives

$$2 = \frac{p^2}{q^2},$$

hence

$$p^2 = 2q^2.$$

Thus p^2 is even, and therefore p itself must be even. Write $p = 2k$ for some integer k. Substituting into the equation $p^2 = 2q^2$ yields

$$(2k)^2 = 2q^2 \quad \Longrightarrow \quad 4k^2 = 2q^2 \quad \Longrightarrow \quad q^2 = 2k^2.$$

Hence q^2 is even, so q is even as well.

We have deduced that both p and q are even, which means they have a common factor 2. This contradicts the assumption that $\gcd(p, q) = 1$.

Therefore our initial assumption that $\sqrt{2}$ is rational is false. It follows that $\sqrt{2}$ is irrational. □

2.4.3 Proofs by Induction

Proof by induction is a technique used to show that a statement holds for all natural numbers. The idea is to prove the claim in two steps: first, show that it is true for an initial value (usually $n = 0$ or $n = 1$); this is called the *base case*.
Second, assume that the statement is true for some arbitrary natural number n, and then prove that it must also be true for $n + 1$ (this is called the *inductive step*.
If both steps are established, the statement holds for all natural numbers.
More formally:

- Assumw $P(1)$ true
- $P(j)$ for $j \leq n$ true
- $P(1)$ and $P(2)$ and ... $P(n)$ true implies $P(n + 1)$ true.

The following is a very simple example:

Theorem 2.4.2:

For every integer $n \geq 1$,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Proof. We prove the statement by mathematical induction on n.

Base case: For $n = 1$,

$$1 = \frac{1 \cdot (1+1)}{2} = \frac{2}{2} = 1,$$

so the formula holds for $n = 1$.

Inductive step: Assume the formula holds for some $n = k \geq 1$; that is,

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}.$$

We must show it then holds for $n = k + 1$. Consider

$$1 + 2 + \cdots + k + (k+1).$$

Using the inductive hypothesis for the sum of the first k terms, this equals

$$\frac{k(k+1)}{2} + (k+1).$$

Factor $k + 1$:

$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2}.$$

Thus

$$1 + 2 + \cdots + k + (k+1) = \frac{(k+1)((k+1)+1)}{2},$$

so the formula holds for $n = k+1$. By induction, the formula is true for all integers $n \geq 1$. $\square$

3 Languages and Grammars

3.1 Introduction

Languages and grammars arise as an attempt to make precise the notion of symbolic structure and transformation, which is at the heart of computation. At the basic level, a language is simply a set of finite strings over a fixed alphabet. This definition abstracts away meaning and focuses only on form. Computation, in turn, can be understood as the process of deciding membership in such a set, transforming strings according to rules, or generating strings with a prescribed structure. In this sense, languages provide the objects of computation, while grammars (and machines) provide the means to describe and manipulate them.

A grammar specifies (with a finite description) how strings in a language can be generated. It does so by means of rules that (recursively) rewrite symbols, starting from an initial symbol. Grammars describe a "generative" view of computation: instead of asking whether a given string is accepted (belongs to a language), they describe how valid strings can be constructed step by step. This perspective looks like a computational process, where complex objects are built from simpler ones according to fixed rules. **The recursive nature of grammars is essential, as it allows finite descriptions to define infinite sets of strings.**

The connection between grammars and computation becomes clear observing that different classes of grammars correspond exactly to different models of computation. In the next chapters we will investigate these models together with the corresponding grammars and languages. In the following we will try to clarify more formally what is intended with languages and grammars.

3.2 Alphabet, Symbols, Words, Languages

Alphabet

An Alphabet is a finite set Σ of symbols.
Alphabets can have the "empty symbol" ϵ as one of their elements.
In general, s is a symbol of Σ if $s \in \Sigma$.

A simple example of alphabet is the *binary alphabet* $\Sigma = \{0, 1\}$.
With the symbols we can construct

Words

A **word** is a finite sequence of symbols chosen from an alphabet.
For example, $w = 011010$ is a string (word) from $\Sigma = \{0, 1\}$.

The **length** of a string corresponds to the number of symbols in it and it is indicated with $|w|$.

It is useful to have a notation for special collections of strings:

Fixed-length alphabets and Kleene's Star

$\Sigma^0 = \{\epsilon\}$
$\Sigma^1 = \Sigma$
$\Sigma^{n+1} = \{xy | x \in \Sigma, y \in \Sigma^n\}$
$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$
$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$ (Kleene's Star)

The Kleene's star is particularly interesting since it defines a set composed by all the words of all the lengths which can be constructed with the symbols at hand.
For example, given the alphabet $\Sigma = \{1, 0\}$:

$\Sigma^0 = \{\epsilon\}$ (so ϵ symbol can be regards as a "word" of length zero).
$\Sigma^2 = \{00, 01, 10, 11\}$
$\Sigma^+ = \{0, 1, 00, 11, 01, 10, 000, 001, ...\}$
$\Sigma^* = \Sigma^+ \cup \Sigma^0 = \{\epsilon, 0, 1, 00, 11, 01, 10, 000, 001, ...\}$

An useful operation on strings is **concatenation**

String Concatenation

Let x, y be two strings. The string xy denotes their *concatenation* which is a copy of x followed by a copy of y.
Concerning the length of the concatenation, $|x| + |y| = |xy|$

Having an alphabet and the possibility to construct strings with it, we can now construct specific sets of strings (like we did before with the powers of Σ) which are called **languages**.

Language

If Σ is an alphabet and $L \subseteq \Sigma^*$, then L is a **language over** Σ.

Σ^* is the set of all the words of all the lengths we can build with the symbols of Σ and language is a certain subset of it. It is like considering the usual latin alphabet $\Sigma_L = \{a, b, c, d, ..., z\}$ with $\Sigma_L^* = \{a, b, .., z, aa, ab, ac, ..., house, car, ...\}$. From all the possible words, we define the "English language" as $L_{Eng} = \{house, car, ...\}$.
We need a way to specify alphabets, and a simple one is through the use of set theory-like definitions.
For example, we can describe the set of all words w with $|w| = 5$ over the binary alphabet starting with '0' and ending with '1' with

$$L = \{w : w = 0x1, x \in \Sigma^3\} \quad .$$

3.3 Language Operations

Having defined languages as sets of words, we can also devise different ways to combine them. Given $L_1, L_2 \subseteq \Sigma^*$:

- **Union:** $L_1 \cup L_2 = \{x \in \Sigma^* | x \in L_1 \vee x \in L_2\}$
- **Intersection:** $L_1 \cap L_2 = \{x \in \Sigma^* | x \in L_1 \wedge x \in L_2\}$
- **Complement:** $\bar{L_1} = \{x \in \Sigma^* | x \notin L_1\}$
- **Concatenation:** $L_1L_2 = \{xy \in \Sigma^* | x \in L_1, y \in L_2\}$
- **Kleene's Star:** $A^* = \bigcup_{i \geq 0} A^i$

3.4 Grammars

Languages are described by **grammars**. A grammar is made schematically by:

- A start symbol (S)
- A set of production rules

Production rules describe allowed substitutions. Consider this example grammar:

$$\begin{array}{rrrr} & S \to NP\,VP & NP \to D\,N & N \to N\,PP \\ & VP \to VP\,PP & VP \to V\,NP & PP \to P\,NP \\ N \to \text{man} & N \to \text{girl} & N \to \text{glasses} & P \to \text{with} \\ D \to \text{the} & NP \to \text{Jack} & NP \to \text{Jodie} & V \to \text{saw} \end{array}$$

Consider now this sequence of rules application:

$$S \Rightarrow NP\,VP \Rightarrow D\,N\;\;NP \Rightarrow \text{the}\,N\,VP \Rightarrow \text{the girl}\,VP$$
$$\Rightarrow \text{the girl}\,V\,NP \Rightarrow \text{the girl talked}\,NP \Rightarrow \text{the girl saw Jack}$$

The rules are able to generate the sentence "the girl saw Jack" (and many others). The parts of the rules which are not referring to other rules (like "man", "Jack" and so on) are called **terminals**. The generation process stops when we have terminals only.

3.5 Problems: From Languages to Computation

We define a **Problem** in the following way:

Problem

A **Decision Problem** is defined as **language acceptance**:
given a string w over the alphabet Σ and a language $L \subseteq \Sigma^*$ the problem is to decide whether $w \in L$ or not.

Equivalently, a problem is a function

$$f : \Sigma^* \to \{0, 1\}$$

where $f(w) = 1$ if $w \in L$ and $f(w) = 0$ otherwise.

The second equivalent definition looks closely like a problem of **computation**: we have to find a "machine", a computer, or a computational model able to implement the decision function f.

Every algorithm can be seen as recognizing a language, and every language represents a collection of inputs sharing a common computational property.

This perspective unifies logic, algorithms, and automata theory (see next chapters) by reducing diverse computational tasks to the single abstract question of whether a string belongs to a language.

3.6 Are there more Programs than Algorithms to solve them?

The answer to the question in the section title is positive: there are indeed more problems than procedures able to solve them. Let us see why.
Because every algorithm has a finite description, we can encode each algorithm as a finite string. Therefore, the set of all algorithms is a subset of Σ^*.
Since Σ^* is countable, the set of all algorithms is also countable and therefore we can list them using indices taken from $\mathbb{N}$:

$$A_1, A_2, A_3, ...$$

We can assume that each algorithm A_i computes at most one decision function $f : \Sigma^* \to \{0, 1\}$.
Let us now consider how many problems there are.
A decision problem answers "yes" or "no" for every input string $w \in \Sigma^*$: this means that a problem induces a total classification of all strings into two classes (yes/no).
Thus, a problem is completely defined by its inputs, that is, by the set of strings it accepts, and therefore by a subset of Σ^*.
Given a problem i, its corresponding function will be $f_i = (f_i(w_1), f_i(w_2), ...)$ for each $w \in \Sigma^*$. We can arrange all the functions (problems) in a table together with all the possible inputs:

	1	2	3	...
f_1	$f_1(w_1)$	$f_1(w_2)$	$f_1(w_3)$	...
f_2	$f_2(w_1)$	$f_2(w_2)$	$f_2(w_3)$	...
...	...	...	...	...

We define now the function

$$g_n(w_n) = 1 - f_n(w_n)$$

which flips the bit (turns 0s in 1s and vice-versa) along the diagonal of the previous table. This means that $g(n)$ will differ from the first row by the first entry, from the second row by the second entry and so on. This means that $g(n)$ does not belong to the list of problems and therefore problems are not countable. This is exactly the same "diagonal" argument of Cantor in proving the uncountability of $\mathbb{R}$.

A shorter equivalent proof is the following. Since every problem corresponds to a subset of strings in Σ^*, the set of all problems is in the power set $\mathcal{P}(\Sigma^*)$.
By **Cantor's theorem** there is no bijection between a set and the corresponding power set (even if it is infinite) and therefore $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$.

This implies that there are more problems than algorithms. A direct consequence is that *there exist problems that cannot be solved algorithmically*. Equivalently, algorithms can describe some subsets of Σ^*, but they can never describe all of them.
This somewhat surprising result was made concrete by A. Turing, who explicitly identified one such problem: the Halting Problem.
The fact that the cardinality of $\mathcal{P}(\Sigma^*)$ is that of the continuum also means that, if we were to "extract" a problem at random from this set, we would pick a problem without an algorithmic solution *almost* surely: non-solvable problems vastly outnumber solvable ones.
This fact is not evident from everyday practice. We tend to believe that most problems are solvable (even if "difficult"), but this impression arises because we focus only on problems we care about. A "large" portion of the problems in $\mathcal{P}(\Sigma^*)$ are of little or no interest to us.

A nice example concerns the real numbers, which are not countable (strictly more than integer numbers). Real numbers can be thought as problems. We can express some real numbers with formulas and formulas are countable: they are our algorithms. For example,

$$\pi = \sqrt{6\sum_{n=1}^{\infty}\frac{1}{n^2}} \quad .$$

and we have an algorithm for obtaining π with (in principle) arbitrary precison. We can adopt the following definition: A real number is computable if there exists an algorithm that, given n, outputs a rational approximation within 2^{-n}. This means that we can compute its digits to arbitrary precision by a finite procedure.
The set of computable real numbers is countable, dense[1] in $\mathbb{R}$ and measure zero.
Since real numbers are uncountable, the set of uncomputable reals numbers must be un-

[1]If $A \subseteq X$, A is dense in X if every open interval in X contains at least one element of A. Intuitively, no matter how much you "zoom in" on X, you aways find elements of A. For the real numbers: $\forall a < b$, $(a, b) \cap A \neq \emptyset$. For example, $\mathbb{Q}$ is dense in $\mathbb{R}$ while $\mathbb{Z}$ is not.

countable: as for the problems, *most* of the real numbers are uncomputable.

3.7 The Chomsky Hierarchy: Introduction

Here we anticipate some of the topics that will be discussed later in these notes. In this section we just mention them to give an idea about what we are going to investigate next. The general idea is that languages can be classifyed according to their generality. We can devise very simple languages which can be accepted by very simple machines and then work our way up to more complex languages. Many questions can be addressed at this point: is this "tower" or hierarchy of languages finite or not? What are the limitations of the various languages? And the characteristics of the corresponding machines able to recognize them? N. Chomsky was the first attempting a classification of the languages, commonly known as **Chomsky Hierarchy**, where languages are divided in **Types**:

- **Typ. 0: Recursively enumerable languages**
  At the top of the hierarchy are Type-0 languages, also called recursively enumerable languages. These are generated by unrestricted grammars, where production rules have essentially no constraints beyond having a nonempty left-hand side. From a computational point of view, these are exactly the languages recognized by Turing machines. This class captures the full notion of effective computation: anything that can be computed at all, in the sense of **Church–Turing**[2], corresponds to a Typ.0 language.

- **Typ. 1: Context-sensitive languages**
  These languages are generated by context-sensitive grammars, where productions cannot decrease the length of the string (with the usual exception for the empty string). **"Context" refers to the surrounding symbols that matter when a rewriting rule is applied.** This means that in these languages a rule is applied also depending from the symbols surrounding a non-terminal (see next section).
  This syntactic restriction reflects a computational limitation: these languages are exactly those recognized by linear bounded automata, which are Turing machines whose tape is restricted to a region linear in the size of the input. Context-sensitive languages are strictly more expressive than context-free ones and can capture dependencies that require multiple interacting constraints, such as $L = \{a^n b^n c^n | n \geq 0\}$. All context-sensitive languages are decidable[3] : membership can always be determined in finite time.

[2]Turing machines and the Church-Turing thesis will be explained later on.

[3]Decidability is a fundamental property of languages and logic systems: it will be explained later when analyzing different languages.

- **Typ. 2: Context-free Languages** The context-free languages (CFLs), are generated by context-free grammars, where each production has a single nonterminal on the left-hand side. These grammars naturally describe hierarchical and nested structures, such as balanced parentheses, arithmetic expressions, and block-structured programming constructs. Computationally, context-free languages are exactly those recognized by push-down automata, which extend finite automata with a stack. The stack provides unbounded memory but only in a last-in, first-out manner, which explains both the power and the limitations of CFLs. Many programming language syntaxes live here, though typically only their "core" structural aspects.

- **Typ. 3: Regular Languages** These are the simples languages and we will see that they will also be accepted by the simplest kind of machines, namely the finite state automata. These machines do not have memory.

Typ. 3 languages are the simplest and are included as subset of Typ. 2 languages and so on, therefore the Chomsky hierarchy is characterized by the following inclusions:

$$\text{Regular} \subset \text{Context-free} \subset \text{Context-sensitive} \subset \text{Recursive Enumerable}$$

What makes this hierarchy interesting is that it puts syntactic restrictions, computational models, and computational power into a single framework.
Grammars describe how strings are built, machines describe how strings are recognized, and the hierarchy describes how changes in structural constraints correspond to changes in computational strength.

3.8 The Chomsky Hierarchy

Here we give the formal definitions of the various classes in the Chomsky hierarchy in terms of grammars. The first definition is also the most general one where basically no restrictions on the rules are applied:

Typ.0: Recursively enumerable languages

A Typ.0 grammar (also called *unrestricted grammar*) is a tuple $\langle N, T, P, S\rangle$ with

- N, T disjoined alphabets (non-terminals and terminals)
- $S \in N$ is the start symbol
- P is a set of production rules of the form $\alpha \to \beta$ with:
  $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$

A language is called a Typ.0 language if and only if it is generated by a Typ.0 grammar.

Typ.1 grammars restrict the rules to substitutions which must be larger than the original string:

Typ.1: Context-sensitive languages

A grammar is Typ.1 if all the production rules $\alpha \to \beta$ have the property:
$|\alpha| \leq |\beta|$
The only exception is $S \to \epsilon$ which is allowed if S does not appear in any right-side of the rule.
This class of grammars generate context-sensitive languages.

Considering again the example seen before: $L = \{a^n b^n c^n | n \geq 0\}$, a set of rules generating this language are:

$$\begin{aligned} S \to abc \quad S \to aabCbc \quad abC \to aabCbC \\ Cb \to bC \quad Cc \to Cc \end{aligned}$$

Typ.2 grammars generate context-free languages:

Typ.2: Context-free languages

A grammar of Typ.2 is such that in every rule $\alpha \to \beta$, $\alpha \in N$. Typ.2 grammars generate context-free languages.

The most restricted class is Typ.3:

Typ.3: Regular languages

Typ.3 grammars generate regular languages and all the rules $\alpha \to \beta$ satisfy:
$\alpha \in N$, $\beta \in T^*$ or $\beta = \beta' X$ with $\beta' \in T^*$ and $X \in N$.

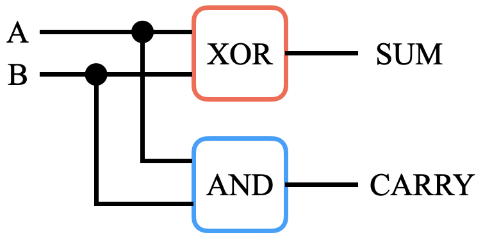
In the next chapters we will investigate which computational models can accept these languages.

4 A first model of computation: Boolean Logic

Since our aim is to explore models of computation and investigare their power and limitations, we start from a very simple one, based on boolean logic. We would like to do calculations just using circuits implementing logic gates. Such computational model can accomodate for example all the arithmetic operations (at the end this is what your pocket calculator does!). Let's see how for example we can add two binary numbers with a circuit.

4.1 The Half-Adder

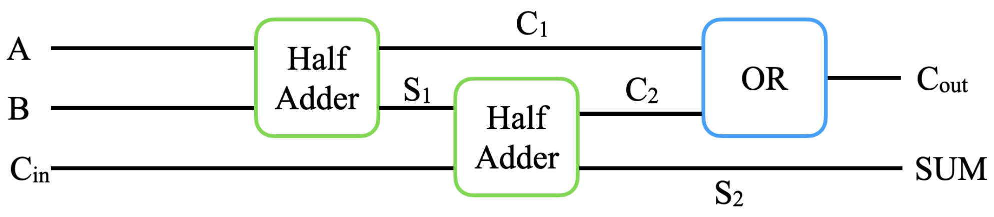
A binary number is a sequence of 0s and 1s and we know the basic results: $0 + 0 = 0$, $0 + 1 = 1$, and $1 + 0 = 1$. The results of these operations look almost similar to the two-input `XOR` gate ($\oplus$). If we consider `XOR` as a gate able to perform sums, we need to consider also the case $1 + 1$ which results in a 0 according to `XOR`, but in reality generates a carry-over (like when in base-10 we add two numbers with a result > 9). The following circuit, called the *half adder* implements the sum of binary digits and recording also the presence of a carry over:



Inputs		**Outputs**	
A	B	S $(A \oplus B)$	$(A \wedge B)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

4.2 The Full-Adder

Having the half-adder which manages the sum and delivers the carry-over bit, we can now define the full-adder as a circuit the performs the addition taking into account the presence of the carry over:‘



This circuit has the bits A and B as input, plus the carryover $C_{in}$from a previous addition: it adds A and B and then adds the result to the previous carryover, producing the result, plus a new carryover C_{out}. It is clear that chaining N full-adders, one can create a circuit able to sum N-bit binary numbers. Having a circuit able to do additions, it is not difficult to imagine that is is possible to devise circuits able to subtract, multiply, and even divide. A "half-subtractor" can be readily realized adding a `NOT` gate to the half-adder transforming the carryover: $A \wedge B \rightarrow \neg A \wedge B$.
Multiplication is more complex than addition or subtraction because it involves repeated addition and shifting, which results in repeated `AND`ing and adding.
Division is even more complex and requires repeated subtraction and shifting.

4.3 Considerations on the Boolean Computational Model

The question is now how powerful is a machine based on the boolean logic model. What a set of logical gates in general does, is to take N bits as input and return M bits as output. Mathematically, they implement a *boolean function*

$$f : \{0,1\}^N \longrightarrow \{0,1\}^M \quad ,$$

where the notation $\{0,1\}^k$ means a sequence of 0s and 1s of length k.
A first observation is that this model has **no memory**: the output depends only on the current inputs and not from some information stored before. Moreover, the input and the outputs have to be of finite and fixed size. Our adder can add only exactly N-bits numbers, but not, for example (N+1)-bits ones (or (N-1)-bits).
The general "complexity" of a boolean machine can be measured for example by the number of the logic gates composing it (circuit size), without any reference to time (or, conversely, the computational time is always proportional to the number of the circuit

size).
Which language is *accepted* by a Boolean machine? The answer seems simple: strings of 0s and 1s, while the alphabet is composed by two symbols only (0,1). Can we improve on that, maybe enriching the language and introduce some kind of memory? The answer is positive, in the sense that more powerful models of computation (and more complex languages) exist.

5 Finite Automata and Regular Languages

Finite automata are a simple model of computation characterized by a limited amount of memory. Surprisingly, they can perform a variety of tasks and recognize languages that are relevant in many applications, particularly in computer science. While they are far from the capabilities of a *real computer*^{TM}, studying them provides a clear and useful framework for understanding the key properties required for more powerful models of computation. There is also a probabilistic couterpart of finite automata, called *Markov Chains*.

5.0.1 Definition of Finite Automata

We first introduce finite automata (FA) with a very simple example: Imagine a coffee vending machine that only dispenses coffee when the correct amount of coins is inserted. The machine accepts 1$ coins only. One needs to insert exactly 2$ to get coffee [1]. If you insert more, the machine returns extra money. We can model this device with a FA. The machine has 3 states:

- S0: No money inserted
- S1: 1$ inserted
- S2: 2$ inserted $\rightarrow$ coffee dispensed

The *alphabet* is constituted by two "letter" only: 0 (no coins) and 1 (1$ coin inserted). We have to define also *transitions* between states:

[1] US dollars do not have 1- or 2-dollar coins, so consider $=Canadian Loonies (or Australian coins)!

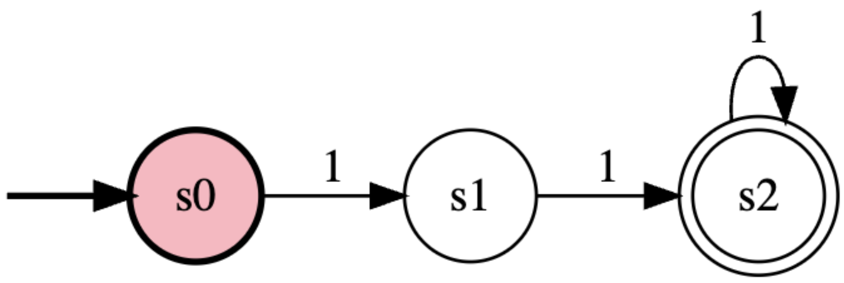


Figure 5.1: Diagram of the "Coffee Vending Machine" Finite State Automaton.

Current State	Input	Next State	Action
S0	1	S1	Accept coin
S1	1	S2	Dispense coffee
S2	1	S2	Return extra coin
S0, S1, S2	0	Current	Do nothing

The whole functioning principle of this FA can be also represented graphically (see Fig. 5.1). In a FA graph, states are represented with circles, while arrows indicate transitions between states. The initial state is the one where an incoming arrow does not come from any other state (this state is colored in Fig. 5.1). The *final state* or **accepting state** is marked with a double circle and in this case corresponds to the state S2, where the coffee is delivered (after inserting two coins).
Now that we have an intuitive understanding of what a finite automaton is, we can provide a formal definition:

Finite Automaton

A finite automaton is a 5-tuple (Q,Σ,δ,q_0,F) where:

1. Q is a finite set called **states**,
2. Σ is a finite set called **alphabet**,
3. δ:Q$\times\Sigma\rightarrow$Q is the **transition function**,
4. $q_0\in$Q is the **start state**,
5. F$\subset$Q is the **set of accepted states**.

Let us consider a slightly more complex FA for understanding the definition in an-

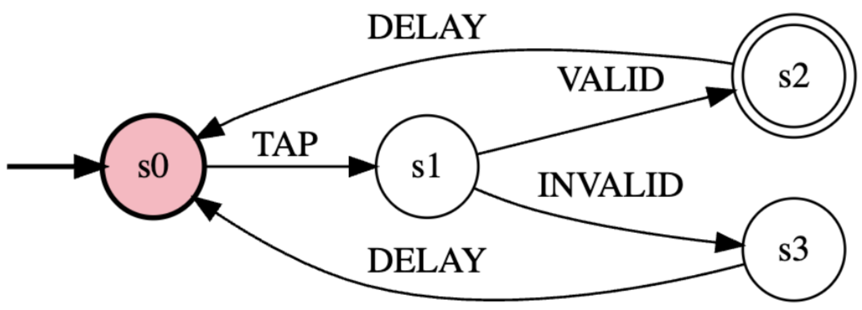


Figure 5.2: Diagram of the "Metro Turnstile with Smart-Card Validation" Finite State Automaton.

other concrete case. Imagine a device at the entrance of the metro in a city where passengers have to present a smart-card for entering. If the money on the card is enough (and the card is *valid*), then the door will open. If money is not enough or the card is expired (is *invalid*), the door will not open. In both cases, after scanning the card, the system must return into the starting state. The diagram of such FA is showed in Fig. 5.2. From the definition,

- The set of states is Q = { s0, s1, s2, s3 }
- The alphabet is $\Sigma = \{$ TAP, VALID, INVALID, DELAY$\}$
- The transition functions can be represented with a table:

Q	Σ	δ:Q$\times\Sigma\rightarrow$Q
S0	TAP	S1
S1	VALID	S2
S2	DELAY	S0
S1	INVALID	S3
S3	DELAY	S0

- The start state is $q_0 = s0$
- The subset of accepted states is $F = \{s0\}$.

5.0.2 Accepted Languages

A natural question about FAs is: *what sequence of strings in the alphabet allows the FA to reach an accepted state?*. With "strings" here we loosely refer to a sequence of elements in the alphabet Σ set.
In general, we say that the **language** L of a FA is the set of all the sequences of alphabet

elements (strings) that the FA accepts. We can state the following definition:

Accepted language

If A is the set of all strings a finite automaton machine M accepts, then A is the **language of the machine M** and we indicate this with:

$$L(M) = A$$

NOTE 1: The word "accepts" is commonly referred to both, strings and languages. To avoid confusion, often "recognizes" is used for languages and "accepts" for strings but hopefully the meaning will be clear from the context.
NOTE 2: A FA can accept in general many strings, but it *recognizes* only one language.

In this preliminary discussion, we see that finite automata are tightly bound to the concept of languages and the further questions that arise are: *which languages are accepted by FA? What are their characteristics? Are there languages that are not accepted by any FA? If not, are there other kinds of automata?*
As a next step, in the next section we will introduce strings and laguages in a more formal way.

5.1 Computation

Now that we have defined FAs and strings and achieved an intuitive understanding about how FAs process strings, we can try to give a formal definition of **computation**:

Finite Automata Computation

Given a FA M=(Q,Σ,δ,q_0,F) and a string $w = w_1 w_2 ... w_n$ where $w_i \in \Sigma$, we say that M **accepts** w if $\exists r_1, r_2, ..., r_n \in Q$ (a sequence of states) such that:

- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, .., n-1$
- $r_n \in F$

Extending to languages, M **recognizes language A** if $A = \{w | M \text{ accepts } w\}$.

The definition follows what we informally saw before: the FA starts from its initial state q_0 and transitions according to the function δ upon the input of string elements and the state where the FA is. The computation finishes if the last state belongs to one of the FA's acceptes states.

NOTE: It is good to keep in mind the two ingredients needed for a "computation": an input symbol and the fact that the "machine" or automaton is in a definite state. The symbol will induce a state change according to the current state. These ingredients will remain present also in more complex machines, as we will see.

5.2 Regular Languages

Among all the possible languages, a particular class is relevant for FAs:

Regular Language

A language is **regular** if it is accepted by a finite automaton.

Regular languages are a very important class that we are going to study in more details.

Example

For understanding how a FA-based computation works in relation to a regular language, let's consider the alphabet $\Sigma = \{a, b\}$ and the following language:

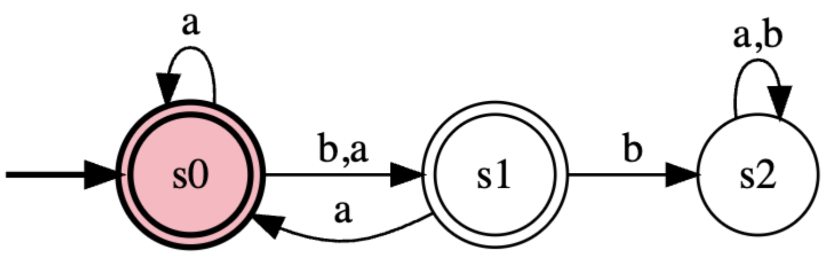
$$L = \{w \in \Sigma^* :' bb' \notin w\} \quad .$$

The last language is composed by all the finite strings made with the letters 'a' and 'b' which do not contain the sub-sequence 'bb'.
The corresponding automaton, must process every string and accepting only if the 'bb' sequence is not there.
The states of such an FA are:

- $s0$: the starting state -> accept - $s1$: the last symbol seen is a 'b' -> accept
- $s2$: a 'bb' is detected -> reject

The transitions among states are:

State	a	b	**Accepting?**
$s0$	$s0$	$s1$	Yes
$s1$	$s0$	$s2$	Yes
$s2$	$s2$	$s2$	No



t

Figure 5.3: The NOT-'bb' accepting finite automaton. The states with a double-circle are the accepting states while the red-filled circle is the initial state.

where we represented the transition function δ in tabular form. The graph corresponsing to this automaton is shown in Fig. 5.3.

Operations on Languages

We can define operations among languages and in particular it would be interesting to identify operations which preserve the language properties. The operations we consider are

- **Union:** $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- **Concatenation:** $A \circ B = \{xy | x \in A \text{ and } y \in B\}$
- **(Kleene) Star**: $A^* = \{x_1 x_2 ... x_k | k \geq 0 \text{ and } x_i \in A \forall i\}$

The Union operator just merges two languages. The *Concatenation* operation attaches a string from a language A in front of a string of language B. We encountered the *Star* operation before while discussing how to build languages from strings: in this case, strings of any length which are combinations of the original strings language are produced.

Example

Let's consider the languages $A = \{0, 1\}$ and $B = \{a, b\}$.
The union is $A \cup B = \{0, 1, a, b\}$.
The concatenation is $A \circ B = \{0a, 0b, 1a, 1b\}$.
The star is: $A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 011, 101, 111, 0000, ...\}$

We are now ready to prove theorems about the operators we just defined.

In particular, we will prove that these operations are *closed*. *Closure* under a certain operation means that the operator takes elements from a specific set and produces as output another member of the same set (Like the "+" operation on natural numbers, for example).

Theorem 5.2.1: Closure under Union

The class of regular languages is closed under the union operation. This means that if A and B are regular languages, also $A \cup B$ is regular.

Proof. The proof will take advantage from finite automata. We know that they can (by definition) process only regular languages, so let's consider two FAs M_1 and M_2 which can accept languages A_1 and A_2, respectively. Their definitions are $M_i = \{Q_i, \Sigma, \delta_i, q_i, F_i\}$ where $i = 1, 2$.
Note that here for simplicity we assume that the alphabet Σ is the same for both FAs.
We show that we can construct explicitly an automaton $M = \{Q, \Sigma, \delta, q, F\}$ able to accept $A_1 \cup A_2$:

1. The new FA has a set of states $Q = \{(r_1, r_2) | r_1 \in Q_1, r_2 \in Q_2\}$: this means that the new set of states is the cartesian product $Q_1 \times Q_2$. We used the cartesian product because M cannot be a "sequential FA" which first processes A_1 and then "rewinds" for processing A_2. Everything must happen in one single pass.

2. The transition function is
$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)) \quad \forall (r_1, r_2) \in Q \quad .\forall a \in \Sigma \quad .$$

3. $q_0 = (q_1, q_2)$

4. $F = \{(r_1, r_2) | r_1 \in F_1, \text{ or } r_2 \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

□

5.3 Regular Expressions

In different programming languages we can specify general strings with the so-called `REGEX` notation. For example, the automaton of Fig. 5.3 accepts strings of the form $a^*(ba^*)^*$ in `REGEX` notation. The notation uses the character '*' for indicating multiple occurences of a sequence ($a^* = aaa...$, $(ab)^* = ababab...$). So the expression $a^*(ba^*)^*$ means: "exactly the strings over {a,b} that contain no occurrence of bb" since:

- a^* means “zero or more a”,
- $(ba^*)^*$ means “b followed by zero or more a”.

Putting the expressions a^* and $(ba^*)^*$ after each other means *concatenation*, sometimes expressed also with '+'. The star operator '*' has precedence over all other operators. Other useful operators are:

- $a+$: one or more 'a'. Very similar to '*' but excludes the empty string ϵ
- $a?$: zero or one repetition of 'a'.
- $[abc]$: matches a, OR b, OR c.
- $[a - d]$: specify a range if a lexycographic order is specified (a OR b OR c OR d).
- $\{n\}$: exactly n repetitions.
- $\{n,\}$: n or more repetitions.
- $\{n, m\}$: between n and m repetitions.
- $\hat{:}$ start of string.
- $: end of string.
- `\b` word boundary.
- `\B` non-word boundary.
- All the combinations are possible, e.g.: $[abc]^*$.

If you know regular expressions from programming languages like `Python` or `Perl`, you might notice that several other operators are missing here. For example, it is not possible to include *backreferences*, *lookahead/behind*, *recursion* or *conditionals*.**All these features would make the language non-regular**, and therefore a FA cannot process them. **In particular, these constructs require memory of unbounded length, which FAs do not possess.** A classic example is the recognition of nested *parentheses*, which requires *recursion* capabilities.

5.4 Relation of Regular Expressions to Regular Languages and Finite Automata

Regular expressions characterize regular languages which are accepted by FAs, so in general:

$$\boxed{\text{Regular expressions} \longleftrightarrow \text{Finite state automata} \longleftrightarrow \text{Regular languages}}$$

We know already that regular languages are defined as the languages accepted by FAs. It remains to clarify the relation between regular expressions and regular languages.

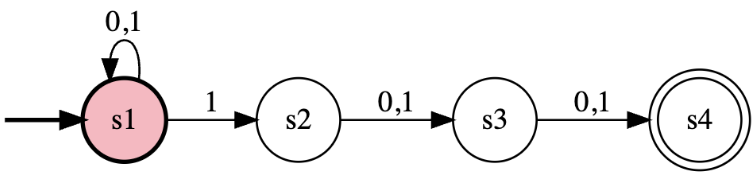


Figure 5.4: Non-Deterministic FA version of the automata accepting strings with '1' in the third position.

The relation is expressed by the following

Theorem 5.4.1: Regularity

A language is regular if and only if some regular expression describes it.

The proof of this theorem and e.g. closure proofs are simpler if we introduce another type od automaton, equivalent to FAs used up to now: the *non-deterministic finite automaton*.

5.5 Non-Deterministic Finite Automata

The FAs we investigated up to now were *deterministic*, in the sense that their functioning is completely determined by their structure and each input leads to an unique output. We introduce here **non-deterministic** finite automata (NFA) which have additional characteristics making them well suitable for different theoretical applications like to proof of theorems regarding FAs.
We introduce NFAs with an example shown in Fig. 5.4. At a first glance, the automaton in the figure seems like all other FAs, but looking closely you can see that from the first node state $s1$, two arrows corresponding to the input '1' are exiting. Such feature cannot be present in FAs: if the input is '1', which path should be choose?
This is exactly the feature of non-deterministic FAs: when a state has more than one exit-arrow, the automaton follows all of them. It is like the automaton clones itself as many times as there are arrows and each clone or copy of the original automaton follows one of them.
An additional feature is the possibility to have arrows marked with the empty string ϵ. When this string is encountered, even without reading an input, the automaton "splits" in two: another copy of itself jumps to the next state pointed by the ϵ-arrow while the "old" copy stays in the present state. In general, more arrows with the empty string symbol can exit a state. We can see these multiple copies of a NFA as a kind of parallel processing or a successive "forking" process that can be represented

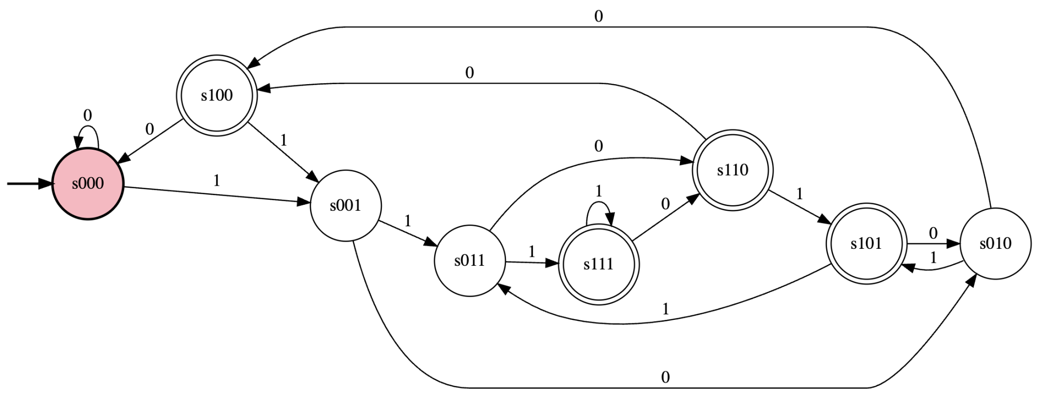


Figure 5.5: Deterministic FA version of the automata accepting strings with '1' in the third position.

with a tree.
What is the NFA in Fig. 5.4 doing is accepting all the strings with a '1' in the third position from the end. How would look a deterministic FA able to do the same thing? Such an automaton is shown in Fig. 5.5: noticing the (huge) difference in complexity and number of states we can start to understand that non-determinism might be useful. The formal definition of a NFA is:

Non-Deterministic Finite Automaton

A Non-Deterministic Finite Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state,
5. $F \subseteq Q$ is the set of accepted states.

Where $\mathcal{P}(Q)$ is the power set of Q and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

This definition is similar to that of a deterministic FA, with the key differences that it takes into account ϵ-transitions and modifies the transition function δ. In this case, the transition function no longer maps the Cartesian product of states and symbols to a single state, but rather to the set of all possible subsets of states $\mathcal{P}(Q)$. This means that from

one state the automaton can move to a set of states: this is the key non-deterministic property of the new function.
With NFA, also the definition of computation is sligthly modified (compare with the definition of FA computation):

Finite Non-deterministic Automata Computation

Given a NFA N=(Q,Σ,δ,q_0,F) and a string $w = w_1 w_2 ... w_n$ where $w_i \in \Sigma$, we say that N **accepts** w if we can write w as $w = y_1 y_2 ... y_m$ where $y_i \in \Sigma_\epsilon \, \forall i$ and $\exists r_1, r_2, ..., r_n \in Q$ (a sequence of states) such that:

- $r_0 = q_0$
- $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i = 0, ..., m-1$
- $r_m \in F$

5.6 Equivalence of FAs and NFAs

The power of NFAs might seem much higher than FAs in terms of computational capabilites so it might sound surprising that actually both recognize the same class of languages! This means that NFAs have only apparently more capabilities with respect to FAs: given a language accepted by an NFA, there is also a FA able to do the same. This amazing fact is stated by the following theorem[2]:

Theorem 5.6.1: Determinization Theorem (Rabin and Scott (1959)

For every non-deterministic finite automaton there is an equivalent deterministic finite automaton.

Proof. The idea behind the proof is the observation that a FA can simulate all the possible paths of a NFA. This can be done keeping track of all the states where the NFA could be after having read a certain part of the input. But how many might these states be? If a NFA has k states, we have to consider all the possible subsets of states what it can reach thanks to the non-deterministic transitions. The set

[2]The theorem was proved by Michael O. Rabin and Dana Scott (1959). This result is part of what earned them the Turing Award in 1976. The orignal paper is: "Finite Automata and Their Decision Problems", IBM Journal of Research and Development (1959).
Stephen Kleene (1956) had already connected regular expressions and finite automata, and his work implicitly contains the determinization idea, but Rabin and Scott provided the first formal proof within automata theory.

of all the possible subsets of a set is the *power set* $\mathcal{P}(A)$ which has cardinality |P(A)|=2^k. This number is the maximum number of states theoretically reachable by the computation (in practice often a much smaller number is reached). We can conclude that for a FA to simulate a NDA with k states needs at most 2^k states. Let's now see more formally how the "simulation" is done.

Let's consider a NFA $N = (Q, \Sigma, \delta, q_0, F)$ recognizing a language A: we will try to construct a FA M=$(Q', \Sigma, \delta', q_0', F')$ which will be also able to recognize A. Initially we consider automanta N without ϵ-arrows for simplicity. Here is the construction:

1. Q' = P(Q) : the states of M are all the possible subsets of Q (i.e. including $\emptyset$ and Q).
2. For R∈Q' and $a \in \Sigma$, $\delta'(R, a) = \{q \in Q | q \in \delta(r, a)$ for some $r \in R\}$.
   This definition of transition function assures us correspondence between states among N and M: when the FA makes a transition, it must end into a state which belongs to the NDA. We can also restate the transition function as
   $$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a) \quad .$$
3. M must start in a state equal to a set containing just the start state of N: $q_0' = \{q_0\}$.
4. $F' = \{R \in Q' | R$ contains an accept state of N$\}$: Accept states must match: M accepts if N is in (at this point of the computation) one of its accepting states.

Considering now the ϵ-arrows, we define E(R) for any state R of M, which is the collection of states that can be reached from R following an ϵ-arrow, including members of R. To be more precise, E(R) contains states reachable with *zero or more* ϵ-arrows. Clearly, $R \subseteq Q$.
This definition allows us to modify the transition function:

$$\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a)) \text{ for some } r \in R\} \quad .$$

Having added the ϵ-transitions, we have still to fix the initial state from which ϵ-arrows can exit:

$$q_0' \longrightarrow E(\{q_0\}) \quad .$$

Having constructed a FA which at each state of the computation enter a state belonging also to the corresponsing NFA completes the proof. □

A corollary, based on the regular language definition and the latter theorem is that a **language is regular if and only if it exists a non-deterministic automaton able to recognize it.**

5.7 Closures

Before we anticipated that regular languages are closed under the union operation (Theorem 5.2.1) but stated that NFAs would be very useful for simplifying closure proofs. In the following we will show how regulare languages are closed under union, concatenation, and star operations.

Theorem 5.7.1: Closure under Union

Regular languages are closed under Union.

Proof. Having two regular languages L_1 and L_2, we want to prove that $L_1 \cup L_2$ is regular. The idea is to define two NFAs N_1 and N_2 accepting the respective languages and then merge them in a single NFA. In this way, we leverage on non-determinism, which helps in choosing after each step which NDA automaton to use.
Let: $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing L_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizing L_2, we construct $N = (Q, \Sigma, \delta, q_0, F)$ recognizing $L_1 \cup L_2$:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$, where q_0 is the start state of N,

2. The accept states are $F = F_1 \cup F_2$: this is key to the union: N must accept if either of the two automata accepts,

3. The transition function is defined such that $\forall q \in Q$ and $\forall a \in \Sigma_\epsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

□

Theorem 5.7.2: Closure under Concatenation

Regular languages are closed under Concatenation

Proof. Let's consider again, as in the previous theorem, two NFAs N_1 and N_2 and construct a NFA N which recognizes $L_1 \cdot L_2$ (the concatenation):

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$
2. q_1 is the start state (like for N_1)
3. The accept states are F_2 (like for N_2)
4. The transition function is defined such that $\forall q \in Q$ and $\forall a \in \Sigma_\epsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

□

Theorem 5.7.3:

Regular languages are closed under the star operation.

The proof is based on the idea of building a new NFA N able to recognize words of the language L_1^* which can be broken into sub-words recognized by N_1 and belonging to L_1. This can be achieved adding ϵ-arrows going from the accept states to the start state.

Proof. We consider $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing L_1 and construct $N = (Q, \Sigma, \delta, q_0, F)$ recognizing L_1^*:

1. Q $= \{q_0\} \cup Q_1$,
2. The state q_0 is a new start state,
3. F $= \{q_0\} \cup F_1$,
4. The transition function is defined such that $\forall q \in Q$ and $\forall a \in \Sigma_\epsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \epsilon \\ \{q_1\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

□

5.8 Regular Expressions: Formalization

In Sec. 5.3 we introduced regular expressions, mostly with a syntax and operators taken from common programming languages. All the listed operators are handy in practical programming situations but they are equivalent to just 3 simpler operations: union, concatenation, and Kleene star. We can thus define[3] what a regular expression (a specific kind of language) is:

Regular Expression

R is a regular expression if R is:

1. $a \in \Sigma$ (with Σ some alphabet),
2. ϵ (the empty string),
3. $\emptyset$ (a language with no strings),
4. $(R_1 \cup R_2)$ where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$ where R_1 and R_2 are regular expressions,
6. (R_1^*) where R_1 is a regular expression.

Let's look at some simple examples showing that the usual `REGEX` expressions in programming languages are equivalent to the three expressions we have listed above:

- Range operator: $[1-3] = 1 \cup 2 \cup 3$
- Substring: $[abc] = a \cup b \cup c$
- Appearence or not: $a? = a \cup \epsilon$
- One or more: $a+ = a \circ a^*$
- Bounded repetition: $a\{3\} = aaa = a^3$
- Between m and n repetition: $a\{2,4\} = aa \cup aaa \cup aaaa$

[3]Note the *inductive* nature of the definition, where for refining the regular expression R we use other regular expressions like R_1 and R_2. This might lead us to suspect a kind of circularity, which is not really there since we can notice that the definition allows us to build larger and larger regular expressions starting from smaller ones.

5.9 Equivalence between FAs and Regular Expressions

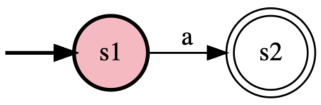
In Sec. 5.4 we anticipated that FAs and regular expressions are equivalent. This means that every regular expression can be converted in a FA and vice versa: this is the content of Theorem 5.4.1 which we are now going to prove.

Proof.

PART1 ($\Rightarrow$) If a language is described by a regular expression, then it is regular (accepted by a FA).

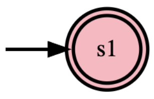
We will convert a regular expression R into a NFA which can recognize it. This is the "simple" direction of the proof, since knowing the definition of regular expression, we closely follow it showing that a NFA can be constructed going over the six parts of the regular expression's definition:

1. $R = a$ for $a \in \Sigma$, therefore $L(R) = \{a\}$. A NFA recognizing this language is readily constructed:

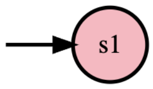
   Here it is interesting to note that an equivalent FA would have been more complex, since we should have taken into account also states for different kind of inputs. The formal definition is: $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$ with $\delta(q_1, a) = \{q_2\}$ and $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \epsilon$, therefore $L(R) = \epsilon$ recognized by:

   which has definition $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ and $\delta(r, b) = \emptyset\ \forall r, b$.

3. $R = \emptyset$ therefore $L(R) = \emptyset$ recognized by:



which has definition: $N = (\{q_1\}, \Sigma, \delta, q, \emptyset)$ and $\delta(r, b) = \emptyset \ \forall r, b$.

4. $R = R_1 \cup R_2$

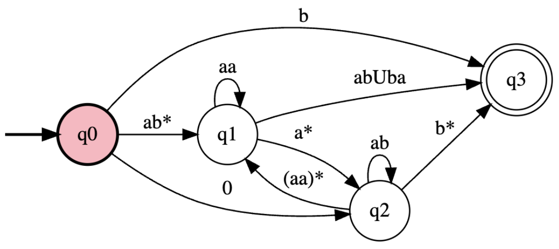
5. $R = R_1 \circ R_2$

6. $R = R^*$

The last three points are direct consequence of the closure theorems proved before.

PART2 ($\Leftarrow$) If a language is regular, then it is described by a regular expression.

For proving this direction of the theorem, we need a procedure for converting an automaton to a regular expression. To this aim, we define one more type of automaton: the *generalized non-deterministic finite automaton*, or GNFA. The GNFA is a NFA which can transition between states *also* reading a sub-string (a sequence of characters) which is a regular expression. **We require that in a GNFA the start state has only outgoing arrows, there is only one single accept state (distinct from the start state), and except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.**
This is an example of GNAF, where transitions are governed by single symbols or regular expressions (**notice that the definition is the same as for a NFA, except for the transition function)**:



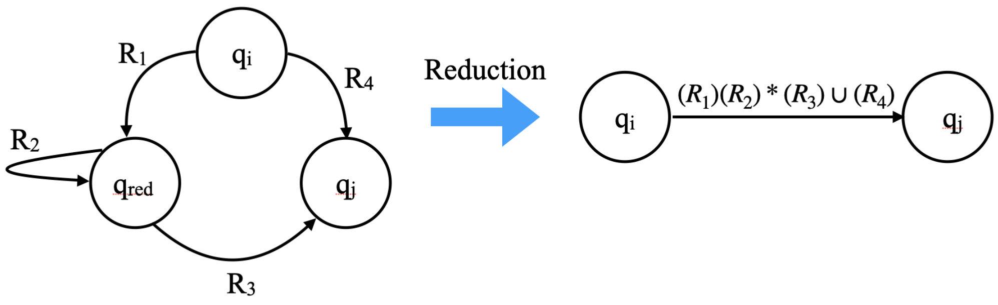
Note that a GNFA can always be converted in a DFA (they are equivalent). Here is the formal definition:

Generalized Nondeterministic Finite Automaton

A GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where:

1. Q is a finite set of states,
2. Σ is the input alphabet,
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow \mathcal{R}$ is the transition function,
4. $\mathcal{R}$ is a collection of regular expressions over Σ. For example if $\delta(q_1, q_j) = R$, the regular expression R is "on the arrow" between the states R_1 and R_2.
5. q_{start} is the start state and q_{accept} is the accept state.

What we have to do now is to convert a GNFA into a regular expression. This is done with a "reduction" process where we convert a GNFA with N states to an equivelent GNFA with N-1 states. The process is iterated until we reach N=2 where a single arrow goes from a start state to the accept state: the transition will be our final regular expression. The reduction idea is depicted as an example in the following figure:



Here, R_1, R_2, and R_3 are some regular expressions and we would like to "eliminate" or reduce the state q_{red}. This is done with regular operations (union, concatenation, star).
Now that we introduced the concepts of GNFA and state reduction for obtaining a final regular expression (the full idea of the proof, basically), we can give its formalization:

First, we consider a FA for some language A and convert it to a GNFA G (adding start and accept states and the additional necessary transitions). After that, we define a **recursive** procedure `CONVERT(G)` which takes G and reduces it to a regular expression R:

`procedure CONVERT(G):`

1. k = number of states of G
2. If $k = 2$ return R (this is known as the "base case" in computer programming languages and prevents infinite recursion or "stack overflow").
3. If $k > 2$, select any state $q_{red} \in Q \neq q_{start}$ or q_{accept} and let the GNAF G'=(Q',Σ,δ',q_{start},q_{accept}) where $Q' = Q - \{q_{red}\}$ and $\forall q_i \in Q' - \{q_{accept}\}$ and $\forall q_j \in Q' - \{q_{start}\}$ let:

   $$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4) \quad ,$$

   for $R_1 = \delta(q_i, q_{red})$, $R_2 = \delta(q_{red}, q_{red})$, $R_4 = \delta(q_i, q_j)$.
4. return `CONVERT(G')` .

□

5.10 Summary for Rugular Languages

Regular expressions, regular languages, and finite state automata are three interrelated concepts in formal language theory that describe patterns and structures in strings over a given alphabet. A regular language is a set of strings that can be recognized or generated by a systematic method, and it is precisely the class of languages that can be expressed using regular expressions. Regular expressions provide a symbolic notation for describing these languages, combining basic operations (concatenation, union, and Kleene star) to generate complex patterns from simpler ones. Finite state automata, on the other hand, are computational models consisting of a finite set of states, transitions between states labeled by symbols from the alphabet, a start state, and one or more accepting states. FAs recognize exactly the set of strings that belong to a regular language: a string is accepted if following its symbols leads the automaton from the start state to an accepting state.
The equivalence among these three notions lies in the fact that any regular expression can be transformed into a finite state automaton that accepts the same language, and conversely, the language recognized by any finite state automaton can be described by a regular expression. This equivalence establishes that regular expressions, finite state automata, and regular languages are just different ways to represent the same class of languages.
On a broader prespective, we can define a regular language from three points of view:

1. **Automata definition:** A language is regular if there exists a finite automaton (FA or NFA) that accepts it.

2. **Definition via regular expressions:** A language is regular if it can be described by a regular expression.

3. **Algebraic definition:** The smallest class of languages over an alphabet that contains the empty set, single-letter languages, and is closed under union, concatenation, and Kleene star.

We started with 1., stating that regular languages are those accepted by FAs or NFAs and then proved that regular expressions are another equivalent way. The way 3. is also possible but we will not investigate it although it should be intuitively viable.

5.11 NON-regular Languages

Up to now we learned what FAs can do but for understanding them better, it is useful to investigate what they can't do. Which languages they cannot accept? These languages must be non-regular but then what are the characteristics of these languages and what are the fundamental reasons why FAs cannot accept them? A typical example of non-regular language is:

$$L = \{0^n 1^n | n \leq 0\} \quad .$$

L is non regular, because it cannot be accepted by a FA. Why? In order to process strings like $0^n 1^n$ a FA must be able to record the occurences e.g. of the zeros, but since the number of states is finite and fixed, this cannot happen for any $n \geq 0$.
This example was fairly straightforward, but in many situations determining regularity is not so simple. For that reason, we will develop a technique for answering the question of whether or not a given language is regular.

5.11.1 The Pumping Lemma

The technique we anticipated before is based on this theorem, known as the *pumping lemma*:

Theorem 5.11.1: (Pumping Lemma)

Given a regular language $A \subseteq \Sigma^*$, there is a number $p \in \mathbb{N}$ (the *pumping length*) such that if a string $s \in A$ has length $|s| \geq p$, then s can be represented by $s = xyz$ where:

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

NOTE: x or z can be the empty string ϵ but not y, given condition 2, which is also important since without it the theorem would be trivially true (it just states that you can break a string in two parts which is true for whatever language you might invent).

Proof. The pumping lemma is fundamentally about the limited memory of a finite automaton, as we hinted at the beginning. A FA has only finitely many states so if you feed it a string that is longer than the number of states, then (by the *pigeonhole principle*[4]) the automaton must visit some state twice while reading the string.
This additional state(s) create a loop in the computation: this corresponds to a *property* of the strings belonging to the regular language.
That loop is induced by some substring of the input (called y here) that the automaton can travel through any number of times. Because looping doesn't prevent the automaton from reaching an accepting state, all these "pumped" strings must still be accepted. In other words, when we run the string through the FA, since s has more symbols that there are states, the automaton must at some point revisit the same state. This means the input can be divided into three parts:
- x: part before the loop,
- y: the loop itself,
- z: the rest of the string.

Because the automaton can go around this loop any number of times and still end in an accepting state, all strings of the form xy^iz must also be in the language. The pumping lemma just says that regular languages are limited by finite memory, so long strings must contain a repeating segment that can be "pumped" (repeated or removed) without leaving the language.
Let's now formalize the proof:

[4]The pidgeon-principle is the name for the trivial fact that if you have N pidgeons but N+k holes ($k > 0$), then one or more must contain more than one pidgeon.

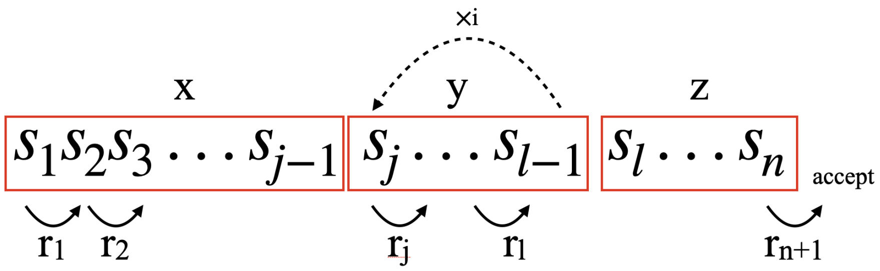


Figure 5.6: Graphical representation of the $s = xy^iz$ string and the FA transitions in the pumping lemma proof.

Let $M = \{Q, \Sigma, \delta, q_1, F\}$ be a FA recognizing a language A and p the number of states of M.
Let $s = s_1s_2...s_n$ be a string of A with $|s| = n \geq p$.
Let $r_1, ..., r_{n+1}$ be a sequence of states that M enters while processing s ($\delta(r_i, s_i) = r_{i+1}$, $1 \leq i \leq n$). The sequence of transitions has length $n + 1 \geq p + 1$.
By the *pidgeon principle*, among the first $p+1$ elements of the sequence, two must be the same state.
Let's call the first "double-state" r_j and the second r_l.
Since r_l happens in the first $p + 1$ steps, $l \leq p + 1$.
Let:

$x = s_1...s_{j-1}$,
$y = s_j...s_{l-1}$,
$z = s_l...s_n$.

The string brings M:
x: from r_1 to r_j
y: from r_j to r_j (goes back to the same state!)
z: from r_j to r_{n+1}

The last state is the accept state, therefore M must accept also xy^iz for $i \geq 0$.
We know that $j \neq l$, so $|y| > 0$ and $l \leq p + 1$ so $|xy| \leq p$: pumping must come from finite memory exhaustion, not from special structure in the tail of th string.
All the conditions of the lemma are satisfied. □

NOTE: The pumping lemma is a **necessary condition** for a language to be regular. It is **not sufficient**. This means that we could have a non-regular language which passes the pumping lemma test.

This is a counterexample:

$$L = \{c^m a^n b^n | n, m \in \mathbb{N}\} \cup (\{a\}^* \circ \{b\}^*)$$

5.11.2 Application of the Pumping Lemma

Let's see how to apply the lemma with the aid of an explicit example. Usually, the lemma is used in proofs by contradiction. Consider the language:

$$L = \{0^n 1^n \mid n > 0\}$$

We know already that it is non-regular, but let's try to prove it formally with the help of the pumping lemma.
Assume, for contradiction, that L is regular. By the pumping lemma, there exists a pumping length $p \geq 1$ such that every string $s \in L$ with $|s| \geq p$ can be written as

$$s = xyz$$

with the following properties:

$$|xy| \leq p, \qquad |y| > 0, \qquad xy^i z \in L \text{ for all } i \geq 0.$$

Choose the string

$$s = 0^p 1^p \in L.$$

Since $|xy| \leq p$, both x and y consist only of zeros:

$$x = 0^a, \qquad y = 0^b \quad \text{with } b = |y| > 0.$$

Now pump (down) with $i = 0$:

$$xy^0 z = xz = 0^{p-b} 1^p.$$

The latter result comes from the fact that putting $i = 0$ removes |y|=b zeros from the string. This string is no longer in L, because it contains fewer zeros than ones, leading to a contradiction.

5.12 Finite Automata Minimization

5.12.1 The Myhill-Nerode Theorem

The Myhill–Nerode Theorem informally says that a language is regular if and only if there are only finitely many distinguishable kinds of prefixes with respect to how they

can be continued to form strings in the language. These "kinds of prefixes" form an equivalence relation that partitions all strings into a finite number of equivalence classes. Each class corresponds to a state in the minimal DFA.
Before enunciating the theorem, we need the definition of equivalence class:

Equivalence Relation and Equivalence Class

An *equivalence relation* on a set A is a binary relation $\sim$ on A satisfying the three properties:

- $a \sim a \, \forall a \in A$ **(reflexivity)**,
- $a \sim b$ implies $b \sim a \, \forall a, b \in A$ **(symmetry)**,
- if $a \sim b$ and $b \sim c$ then $a \sim c \, \forall a, b, c \in X$ **(transitivity)**.

The **equivalence class** of an element a is defined as

$$[a] = \{ x \in A : a \sim x \}.$$

Simple example: congruence over the integers: $a \sim b \Leftrightarrow a = b(mod 3)$. These are all the numbers with the same reminder when divided by 3.

Now the question we would like to ask is: *how many states should a FA at least have for accepting a language* $L \subset \Sigma^*$? The problem is then finding this lower bound on the number of states. Let's start by defining an equivalence relation:

Nerode Equivalence Relation and Index

The **Nerode relation** is defined as

$$x \sim_L y \Leftrightarrow \forall w \in \Sigma^* : xw \in L \leftrightarrow yw \in L.$$

This relation is an equivalence relation and defines the equivalence class:

$$[x]_L = \{y \in \Sigma^* : y \sim_L x\}$$

This class groups all the prefixes which added to the strings of the language generate new strings still belonging to the language. A natural question is *how many are these classes*? Therefore we define also the

Index of L: Index(L) = (number of equivalence classes with respect to $\sim_L$ in Σ^*).

We are now ready to enunciate the

Theorem 5.12.1: (Myhill-Nerode)

Let $M = (Q, \Sigma, \delta, q_0, F)$ a FA which accepts the language L ($\mathcal{L}(M) = L$ (therefore L is regular), then

$|Q| \geq Index(L)$ (and $Index(L) \leq \infty$).

This means that the number of accepting states must be at least equal or larger than the number of the Nerode equivalence classes. In this sense, this theorem represents a more precise tool for proving non-regularity than the Pumping Lemma.
It is at the same time:

- A characterization of regular languages.
- A method to prove non-regularity.
- A recipe to construct minimal FAs (see next).

Let's see now the proof:

Proof. **If L is regular** $\Rightarrow Index(L) < \infty$
Assuming regularity (thus accepted by M), we define $x \sim_M y$ if M reaches the same state after reading x and y.
Clearly, if two strings end in the same state, appending any string w leads to the same acceptance result.
The number of equivalence classes is therefore $Index(L) \leq |Q| \leq \infty$. The idea is that M "collapses" all strings with the same behavior into a single state. That's exactly what $\sim$ does.

If $Index(L) < \infty \Rightarrow$ **L is regular**
Let's suppose L has finitely many classes $C_1, C_2, ..., C_n$. We can construct a FA:
- with one state per class $Q = \{C_1, C_2, ..\}$
- a start state containing the empty string ϵ
- a transition function $\delta(C_1, a)$ going into a class with all the xa strings where $x \in C_i$.
- accepting states are classes C_i containing at least one string in L.

By construction, such automaton accepts exactly L. The idea is that:
- Each state represents "all strings that behave the same for the rest of the input."
- Transitions are consistent because equivalence classes capture exactly the "future behavior."

□

An **example** might help clarifying how the theorem works. Suppose we consider the language:

$$L = \{w \in \{0,1\} : \text{w ends with } 01\}$$

The corresponding (finite) equivalence classes are:

Equivalence Class	**Representative Strings**
C_0	Strings that do not end with 0 or 1, i.e., empty string ϵ
C_1	Strings ending with 0 but not 01
C_2	Strings ending with 01 (accepting class)

therefore $Index(L) = 3$. These classes summarize all the possible strings in the language which can reach an accept state with or without appending another string [5].
Summarizing, the theorem provides a deep connection between the behavior of strings in a language and the structure of finite automata. It tells that if we can classify all strings according to how they behave when we append any continuation, then the language is regular precisely when this classification produces only finitely many distinct behaviors. Each distinct behavior corresponds to a state in a FA.
Philosophically, the theorem reveals that regular languages are fundamentally about distinguishable patterns of "future possibilities". A string is not important for what it is, but for what it implies about what can come next.
Two strings that cannot be told apart by any possible continuation are equivalent and can be represented by a single state. This captures the **minimal necessary memory** a machine needs to recognize a regular language.
In other words, the Nerode–Myhill theorem formalizes the intuition that regular languages are exactly those for which a **finite amount of information about the past suffices to predict membership in the language for any possible future input.** Furthermore, it provides both a criterion for regularity and a construction of the minimal automaton (see next). The theorem is a bridge between the algebraic perspective of equivalence classes and the computational perspective of the automata's states.

[5] You might think that a class is missing: the strings ending with 1 but not '01' but there is no distinct behavior between "start" and "last symbol 1". The language only requires you to remember how close you are to seeing '01' at the end. After reading any string, the only thing that matters for determining acceptance of future inputs is: What suffix do I currently end with, so that I might finish with '01'? There are only three meaningful situations you can be in: 1) the end is already '01', 2) the end is '0' so you are one step away from acceptance, 3) the end is something that is not helpful, e.g., the last symbol is 1 or you're at the start. So this language only requires remembering whether we have just seen '01', whether we have just seen '0', or whether we have seen something else (1 or the start), so these three situations correspond to the three Nerode classes.

5.12.2 How to minimize a Finite Automaton: Hopcroft's Algorithm

The previous theorem shows the path towards constructing the "smallest" FA (in terms of number of states) that can accept a certain language. Given a FA, there may be redundant states, i.e. states that behave identically for all future inputs. The goal of **Hopcroft's algorithm**[6] is to merge such equivalent states efficiently, producing the minimal FA. The key ideas of the method are:

1. Initially, the set of states is partitioned into two groups: accepting and non-accepting states. Any accepting state cannot be equivalent to a non-accepting state.

2. The algorithm repeatedly checks for each input symbol whether states within the same group behave differently. If two states in the same group transition under the same symbol to states in different groups, they are *distinguishable* and are split into separate groups.

3. This process is iterated until no further splits are possible. At this point, each group contains states that are equivalent in the sense that no string can distinguish between them.

4. Each final group of equivalent states is then collapsed into a single state in the minimal DFA.

The algorithm implements the idea as Nerode equivalence: two states are equivalent if and only if they behave identically for all possible continuations.
The algorithm repeatedly refines partitions based on future behavior. If two states behave differently for some continuation, they are splitted. Once no more splits are possible, every group is a set of equivalent states. Hopcroft's idea is always to split the **smaller** subset and track it in a worklist.
Why splitting the smaller subsets is faster?
Every time you process a group in the worklist, you have to consider all input symbols and all other groups to see if they need to be split. Smaller subsets affect fewer states, so splitting and processing them is faster. This ensures the algorithm runs in $O(n \log n)$ time, which is faster than simply compare every pair of states.
Another way to see this intuitively is:

- Imagine we discover a small distinction on a large block of states.
- If we focus on the small new subset, we can quickly resolve all the consequences of

[6]John Edward Hopcroft (1939), *A $n \log n$ algorithm for minimizing the states in a finite automaton*, Proceedings of an International Symposium on the Theory of Machines and Computations Held at Technion in Haifa, Israel, on August 16–19, 1971, Pages 189-196.

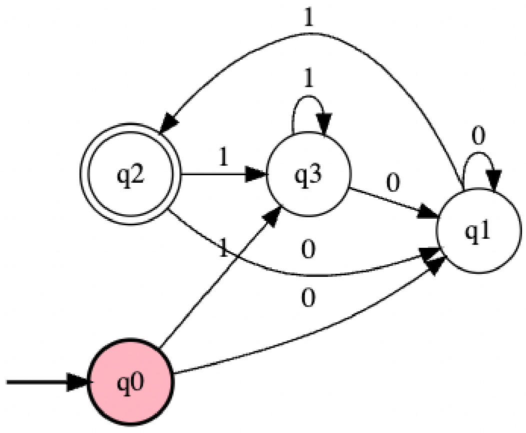


Figure 5.7: Finite automaton accepting strings ending with '01'.

that distinction without affecting the larger group.
- The larger group automatically stays correct, because you only split when necessary.

It is like trying to sort a deck of cards: it is faster resolving inconsitencies in small piles instead of in larger ones!

EXAMPLE:
Let's see how it might work on the previous example for the language where words end in '01' (see Fig. 5.7):

- Start with two partitions: Accepting (01) and Non-accepting (ϵ, 0, 1).
- Check transitions when the inputs are 0 and 1 for the non-accepting group:
    - Some states go to the accepting state (01) appending 1, others do not.
    - Split them accordingly.
- Iterate until each partition contains states that cannot be distinguished by any string (they are Nerode-equivalent classes).
- Each final group is one state in the minimal FA.

Let's try to follow the algorithm step-by-step referring to Fig. 5.7:
STEP 1:
First partition (accepting and not): $P = \{\{q_2\}, \{q_0, q_1, q_3\}\}$.
Initialize the worklist with the smallest set: $W = \{\{q_2\}\}$.

STEP 2:
Following the worklist, we consider q_2: it can be accessed only with the symbol '1' from

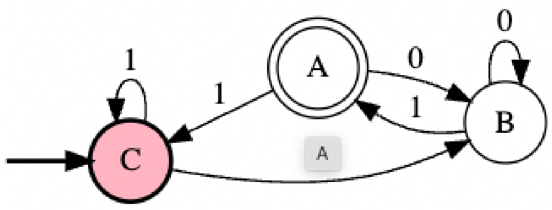


Figure 5.8: Hopcroft-reduced finite automaton accepting strings ending with '01'. The new states represent the equivalence classes:

q_1 thus we split the big block further:
$\{q_0, q_1, q_3\} \longrightarrow \{q_1\}, \{q_0, q_3\}$.

This updates the partition to:
$P = \{\{q_2\}, \{q_1\}, \{q_0, q_3\}\}$
and the work list is now: $W = \{\{q_1\}\}$ (always inserting the smallest set).

STEP 3:
Processing the worklist content, the states from which we can reach q_1 with the symbol '0' are q_0, q_1, q_2, q_3 (all of them). This causes no splitting: P stays the same and the worklist becomes empty.

STEP 4:
There are no more splits available ($W = \emptyset$): the algorithm ends. The algorithm guarantees that when the worklist is empty, all the blocks in P are "stable".

The final reduced FA is in Fig. 5.8). The number of states correspond to the number of Nerode classes.

5.12.3 The Table-Filling Algorithm

This graphical method allows to identify equivalent states in a FA and it is based on a 2-entry table to fill according to how you can reach the accept state. A practical example is the simplest way to understand the method. Let us consider again the 01-ending FA seen before. Suppose it is constructed with four states: q_0 (start), q_1 (last symbol is '0'), q_2 (last is '01': accept), q_3 (last is '1'). We can build the following "triangular" table connecting all the states:

	q_1	q_2	q_3
q_0			
q_1			
q_2			

The gray-shaded entries are not used (they connect states with themselves). The first step is to mark (with an X) all the states connected with the accepting state (q_2):

	q_1	q_2	q_3
q_0		X	
q_1		X	
q_2			X

Now we *propagate* using the transition function. For example, let's check the (q_1, q_3) couple: we see that on imput '1', $q_1 \to q_2$ and $q_3 \to q_3$. At least one transition connects to the accept state, so we mark the (q_1, q_3) entry. After checking all the squares, we end up with

	q_1	q_2	q_3
q_0	X	X	
q_1		X	X
q_2			X

Only the entry (q_0, q_3) is not marked: this means that the two states are equivalent, as we descovered already before. Being in the start state q_0 or having a '1' as last symbol is equivalent: in both cases we need to wait for something else in order to move to the accept state.

In a more formal way, we can summarize the algorithm as:

1. Build the table with the couples $(q_0, q_1), (q_0, q_2), ..., (q_{n-2}, q_{n-1})$ using all the n states.

2. If **exactly one** of the two states is accepting, the pair is distinguishable, so mark it:
$$\text{If } p \in F, q \notin F, \text{ mark } (p, q)$$

3. For each unmarked pair (p, q), and each input symbol a, look at the pair
$$(\delta(p, a), \delta(q, a))$$
If that pair is marked, then (p,q) must also be marked, because you can distinguish them by reading a and then a distinguishing suffix. Mark this, and repeat until no new marks appear.

4. Pairs that remain unmarked correspond to Nerode-equivalent states.

5.13 Summary

- We defined a simple form of computation based on limited-memory Deterministic Finite State Automata
- We defined Regular Languages: they are exactly the languages accepted by Deterministic Finite State Automata
- Non-deterministic finite state automata accept the same languages so they are equivalent to the deterministic version.
- Regular expressions are equivalent to regular languages and are therefore accepted by finite state automata.
- The Pumping Lemma and the Myhille-Nerode theorem are tools for checking for regularity.
- Two strings are Nerode-equivalent if the future cannot tell them apart or equivalently, if, no matter what you append to them, either both become accepted or both become rejected. If finishing the string with any continuation leads both to the same yes/no answer, then such strings are “the same” as far as the language is concerned. The past may be different, but the remaining behaviour is identical so the FA does not need two separate states to represent them (which leads to the minimization procedures).

6 Context-Free Languages and Push-Down Automata

6.1 Introduction

In the late 1950s and early 1960s, researchers were trying to understand what it meant for a string of symbols to belong to a "language" in a rigorous sense, particularly in the context of human languages and programming languages. **(Avram) Noam Chomsky (1928-)**, a linguist, was one of the first to formalize these ideas systematically. He introduced a hierarchy of grammars and languages (Sec. 3.7 and 3.8), with the simplest being regular languages, which could be described by finite automata and regular expressions, and a more expressive class called **context-free languages**.

The term *"context-free"* reflects the idea that the rules used to generate strings in these languages do not depend on the surrounding symbols or "context." In other words, each *production rule* replaces a single symbol with a string of symbols independently of where that symbol appears. This represented a conceptual shift because it allowed grammars to capture **nested, recursive structures** like balanced parentheses, arithmetic expressions, or subordinate clauses in natural language. This could not be achieved with regular grammars.

The motivation for studying context-free grammars came from both linguistics and computer science. Chomsky was interested in describing the structure of human languages in a precise, mathematical way, At the same time of Chomsky's research, the first programming languages were designed and needed a formal method to specify their syntax unambiguously. Context-free grammars fit this purpose as they are expressive enough to capture recursive structures of codes, but are also simple enough to allow efficient parsing algorithms, key to compilers.

6.2 A First Example

We consider here a first example for context-free grammars and show that they are more powerful than regular languages, since they can generate a larger class of languages. To this aim we use a language which FAs cannot accept in general, i.e. the one made of *palindromes*. Palindromes are strings which can be read from both sides, like '11011' or 'anna'.

A finite automaton has only a finite amount of memory, encoded in its finite set of states. When it reads an input word, it processes the symbols one by one from left to right and, once a symbol has been read, it cannot go back to it or store an unbounded amount of information about it. To recognize palindromes, however, a FA needs to remember the entire first half of the word so that it can compare it with the second half read later, in reverse order. Since palindromes can be arbitrarily long, this would require remembering an arbitrarily long prefix of the input and a FA cannot do this.

A context-free grammar can generate palindromes because, unlike a FA, it can use a **recursive structure**. The key idea is that palindromes are self-similar: if you remove the first and last symbol (which must be equal), what remains is again a palindrome.

Consider this set of rules for words made only by '0' or '1' symbols:

$$\begin{aligned} 1:&\ S \to 0S0 \\ 2:&\ S \to 1S1 \\ 3:&\ S \to 0 \\ 4:&\ S \to 1 \\ 5:&\ S \to \varepsilon \end{aligned}$$

Rules 3,4,5 say that a string S can be made by the symbols 0,1, or the empty string. Rules 1 and 2 are the *recursive* structures we were mentioning before: they can generate a new string S starting from the previous one appending the proper symbols as pre- and suffixes. The list of rules generates all the palindromes over the $\{0, 1\}$ alphabet.

This is an example of a **grammar** and rules 1-5 are **substitution rules**.
Each rule of the grammar is comprised by a symbol and a string separated by an arrow. The symbol is called a **variable**. The string consists of variables and other symbols called **terminals**. The variable symbols often are represented by capital letters (S in this case). The terminals are the alphabet symbols (0,1 in this case) and often are represented by lowercase letters, numbers, or special symbols (like ϵ). One specific variable is the **start variable** and usually it is the topmost, left one (S in this case).

6.3 String Generation

Let's suppose we would like to see if the previously defined palindrome grammar can generate the word: '00100'. We have to start from the start variable S, then pick the correct rules from the list:

$$S \xrightarrow{1} 0S0 \xrightarrow{1} 00S00 \xrightarrow{4} 00100 \tag{6.1}$$

Over the arrows we indicated which rule we applied for generating the next string. The last rule substitutes S with '1' and the process is complete.
NOTE: A natural question arises: how difficult it is to check if a "long" word is generated by a grammar? Naively, the problem could become quickly difficult as the length of the word $n = |w|$ grows. It turns out that this verification can be carried out in a time scaling like $O(n^3)$ or even shorter in case of special word classes.
To what extent can context-free grammars generate a "natural language" like English? Let's take Latin as an example and consider the following set of substitution rules (where for brevity we introduce the '|' symbol as an 'OR'):

$$S \to NP\,VP$$

$$\begin{aligned} NP &\to N \\ NP &\to \mathit{Det}\ N \\ NP &\to \mathit{Adj}\ N \\ NP &\to \mathit{Det}\ \mathit{Adj}\ N \end{aligned}$$

$$\begin{aligned} VP &\to V \\ VP &\to NP\,V \end{aligned}$$

$$\begin{aligned} \mathit{Det} &\to \textit{illa} \mid \textit{hic} \\ \mathit{Adj} &\to \textit{bonus} \mid \textit{magnus} \mid \textit{parvus} \\ N &\to \textit{puella} \mid \textit{servus} \mid \textit{agricola} \\ V &\to \textit{amat} \mid \textit{videt} \mid \textit{laudat} \end{aligned}$$

based on the non-terminals:

S = Sentence
NP = Noun Phrase (something behaving like a noun)
VP = Verb Phrase (somethinng behaving like a verb)
N = Noun
V = Verb
Adj = Adjective
Det = Determiner

Try to verify that the following sentences can be constructed with the defined grammar:

Puella amat
servus puellam videt (Here we have to ignore cases like puella->puellam)
bonus agricola servum laudat

It is interesting to note that this grammar does not handle *cases*. A context-free grammar can do it but at the price of complicating the rules unnaturally. Linguistically, this is why more expressive formalisms (feature grammars, unification grammars, dependency grammars) are often preferred: they allow you to say "this noun phrase is accusative" as a feature, instead of hard-coding it into many separate symbols.

6.4 Context-Free Grammars

After the latter examples, we can now define formally a context-free grammar:

Context-Free Grammar

A **context-free grammar** is a 4-tuple (V, Σ, R, S), where:

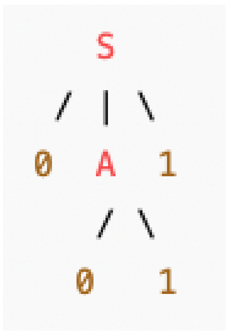
1. V is a finite set of **variables**.
2. Σ is a finite set of **terminals** and $\Sigma \cap V = \emptyset$.
3. R is a finite set of **rules**. Each rule is composed by a variable and a string of variables and terminals.
4. $S \in V$ is the start variable.

6.5 Parse Trees

Derivations such as in 6.1 can be represented graphically with so-called **parse trees**. The case in 6.1 is particularly simple as it involves only one variable (S). Let's consider a slightly more complex grammar with two variables with the following rules:
$S \rightarrow 0A1|a$

$A \to 0A1|01$

This grammar generates strings with balanced numbers of 0s and 1s, with at least one pair, and also allows the single terminal a. For example, the word $w = 0011$ belongs to the grammar, since: $S \to 0A1 \to 0(01)1$. The corresponsing *parse tree* is:



```
    S
  / | \
 0  A  1
   / \
  0   1
```

From the previous diagram, it is clear how a parse tree is constructed: it follows (from top-down) the successive applications of the grammar rules, representing pictorially the various substitutions:

1. Each interior tree node represents a variable,
2. Each leaf represents a variable or a terminal,
3. If the terminal is ϵ, it must be the only child node of the parent one,
4. If a node relative to a variable A has children $A_1, A_2, ..., A_n$, this correspond to the production rule $A \to A_1, A_2, ..., A_n$.

6.6 Example: Parenthesized Arithmetic Expressions

Often we write code containing arithmetic or algebraic expressions, which are then read by an interpreter or a compiler during a process called *parsing*. Let's consider a simple context-free grammar describing such expressions. We define a grammar $G = \{V, \Sigma, R, S\}$ where:

- the variables are: $V = \{Expr, Term, Factor\}$
- the terminals are: $\Sigma = \{a, +, \times, (,)\}$
- the rules are:

1. $Expr \rightarrow Expr + Term|Term$
2. $Term \rightarrow Term * Factor|Factor$
3. $Factor \rightarrow (Expr)|a$

- the start variable is $S = Expr$

As an exercise, try to prove that $a + a \times a$ and $(a + a) \times a$ are generated by the defined grammar and draw the corresponsing parse trees.
Notice how parentheses are correctly used for defining the operations ordering.

6.7 Ambiguity

The definition of a grammar can sometimes generate the same word (or set of words) in many different ways. This is equivalent to say that for at least one string there is more than one parse tree. This phaenomenon is called **ambiguity** and the corresponding grammar is **ambiguous**. If the ambiguity cannot be removed, the grammar is called **inherently ambiguous**.
Two different derivations of a string can differer just in the order of substitutions applied: this is not a structural difference. In order to focus only on the structure of the string, we introduce the following definition:

Leftmost Derivation

A contect-free grammar G has **leftmost derivation** if at every step the leftmost variable is the one considered for substitution.
This (arbitrary) choice removes non-structural ambiguities.

Now we are ready for a consistent definition of ambiguity:

Ambiguous Context-Free Grammar

A string w in a grammar G is derived ambiguously if it has more than one leftmost derivation. G is called *ambiguous* if at least one string is derived ambiguously.

Considering again arithmetic expressions, a grammar with the rule $A \rightarrow A+A|A*A$ is ambiguous, since the expression $A + A * A$ has two derivations:

1) $A \rightarrow A + A \rightarrow A + A * A$
2) $A \rightarrow A * A \rightarrow A + A * A$

Following two different paths we arrive at the same final expression, but the difference is really relevant: in one case we apply the sum first, and in the other one the multiplication. If this rule is part of a computer compiler, we could incur in a wrong priority order of the arithmetic operations.

6.8 Chomsky Normal Form

A grammar can be rather complex and in general it is convenient writing it in the simplest form which follows a kind of standard. The **Chomsky Normal Form** realizes this idea [1]:

Chomsky Normal Form

A context-free grammar G is in **Chomsky Normal Form** (CNF) if every rule is defined in the form:

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is a terminal, A,B,C variables, and B,C are not the start variable.
The rule $S \rightarrow \epsilon$ where S is the start variable is admitted.

The question now is if this form really generates any context-free language. The positive answer is given by the following theorem:

Theorem 6.8.1:

Any context-free language is generated by a context-free grammar in Chomsky Normal Form.

Proof. For proving the theorem we can devise an algorithm which converts any grammar in CNF. In order to do this, we need procedures for substituting grammar rules which do not conform to CNF. These are the steps needed for the conversion:

1. Add a new start variable S_0 together with the rule $S_0 \rightarrow S$: this is needed so we are sure that the start variable is on the left side of a rule, according to the CNF definition (remember: in $A \rightarrow BC$ rules, B and C cannot be start variables).

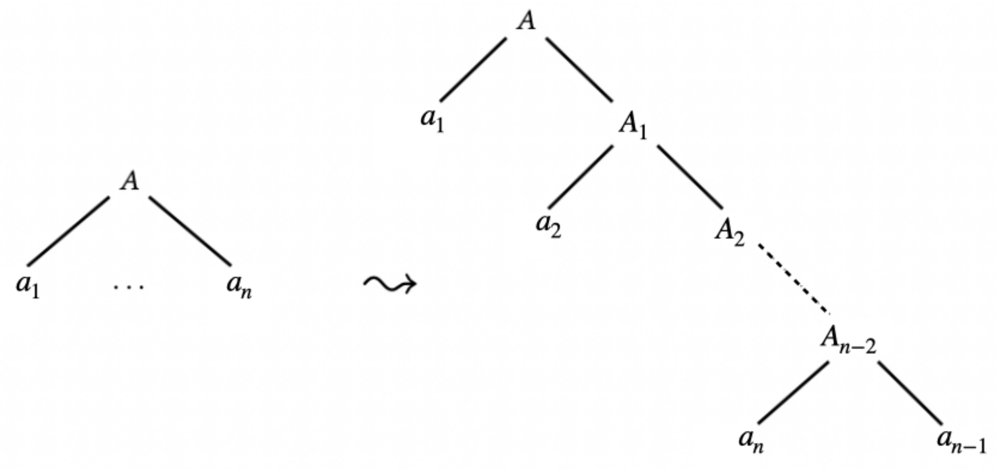
2. Eliminate ϵ-production rules ($A \rightarrow \epsilon$, where A is not the start variable). Then for each occurence of A in the right-side of a rule, we add a new rule

[1]For completeness, another standard form for context-free grammars is the **Greibach Form**.

without A (e.g. for $R \to aAb$ we add $R \to ab$). We have to repeat this addition process for each occurrence of A.
For example, if we have $R \to aAbAc$, we have to add $R \to abAc$, $R \to aAbv$ and $R \to abc$. If we have a rule of type $R \to A$, we add $R \to \epsilon$, unless we removed this rule before with the already stated procedures (a, b are generic strings of variables and terminals).

3. Eliminate all the unary production rules like $A \to B$. Since we remove them, we have to add a rule $B \to a$ whenever $B \to a$ appears (a is a generic string of variables and terminals).

4. Change all the remaining rules to the proper Chomsky form: given variables/terminals combinations $a_1, a_2, a_3, ..., a_n$ with $n > 2$, replace rules like $A \to a_1, a_2, ..., a_n$ with $A \to a_1A_1$, $A_1 \to a_2A_2$,...,$A_{n-2} \to a_{n-1}a_n$. A pictorial representation of the substitution process is:



In the $n = 2$ case, any terminal a_i in the rules listed in this step is substituted with a new variable B_i and the rule $B_i \to a_i$ is added.

□

6.9 The Cocke-Young-Kasami (CYK) Algorithm

A key problem is understanding if a specific word w can be generated by a CFG G. Formally, we ask if $w \in L(G)$ is true or not. In the case of regular languages, we can verify this in O(|w|) time running the word into the corresponding FA and look if it accepts or not.
In the case of a CFG, strings can be generated in many hierarchical ways. The difficulty

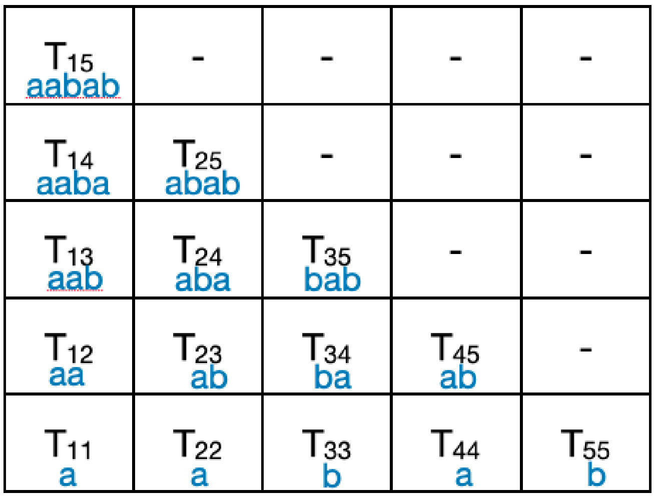
is that you don't know where the grammar "splits" the string. For example, if a rule is $A \to BC$ you do not know in advance "how much" of the string w comes from B and how much from C.
The CYK Algorithm solves this problem using the Chomsky Normal Form by:

- considering all substrings of w,
- computing which nonterminals can generate each substring,
- building the answer bottom-up, from short substrings to longer ones.

Although the procedure seems combinatorial, the running time of the algorithm scales like $O(|w|^3)$: this result is tied to the third point of the latter list, i.e. the bottom-up construction which efficiently re-uses already calculated information. Such procedure is typically known in Computer Science as **dynamic programming**.
A dynamic programming solution of this problem requires the grammar to be expressed in Chomsky Normal form, since it gives rise to a binary-tree structure of the rules (as we saw in the latter figure).

Let's start describing the theorem formally and clarify it with a concrete example later.
CYG Algorithm:

1. Consider a CFG $G = \{V, \Sigma, R, S\}$ reduced to CFN.
2. Let $w = a_1 a_2 ... a_n$.
3. Define a table (matrix) of **sets** of variables: $T_{ij} \subseteq V$.
   The meaning of the entries is: $A \in T_{ij}$ iff the non-terminal A can generate the substring $a_i a_{i+1} .. a_j$. The matrix will look like shown in Fig. 6.1 (e.g. for |w|=5 where w=aabab and $\Sigma = \{a, b\}$):
   The table entries are T_{ij} while below, in blue, the corresponding substring $a_i a_{i+1} .. a_j$ is reported.
4. Initialization of the matrix: the bottom row ($T_{11}, T_{22}, ..$) is made by strings starting at position i and ending at i, therefore they are composed by only one non-terminal (a or b in the example). We are using the rules of the form $A \to a$ in the CNF and we have to place the "head" of the rule (A for $A \to a$ if the non-terminal is a and so on) in the corresponsing table entry. In this way, we fill the bottom row according to the Chomsky rules generating the single non-terminals of the word.
5. Now that the T_{ii} base-case is done, we proceed by induction (this is the dynamic programming step).
   We want now to calculate the entry T_{ij} ($i < j$) which is in the row $j - i + 1$ and we know already the content of the row below (we know how to calculate the



T_{15} aabab	-	-	-	-
T_{14} aaba	T_{25} abab	-	-	-
T_{13} aab	T_{24} aba	T_{35} bab	-	-
T_{12} aa	T_{23} ab	T_{34} ba	T_{45} ab	-
T_{11} a	T_{22} a	T_{33} b	T_{44} a	T_{55} b

Figure 6.1: Matrix filled in the steps of the CYK algorithm. The matrix elements are sets of rule heads.

base-case and we work our way up), i.e. we know already what happened with words shorter than $j - i + 1$. This means also that we know what happened with correct pre- and suffixes before. Since any derivation of our present word derivation $A \to a_i a_{i+1}..a_j$ must start from a rule of the form $A \to BC$ according to the CNF, then
- B derives some prefix: $B \to a_i a_{i+1}..a_k$ for $k < j$ and
- C derives some suffix $C \to a_{k+1} a_{k+2}..a_j$.
Therefore, for having the variable A in the entry T_{ij}, we have to find B,C,k such that:
- $i \leq k \leq j$
- $B \in T_{ik}$
- $C \in T_{k+1,j}$
- $A \to BC$ is a production rule of the grammar G.
This means that for A entering T_{ij}, we have to run over the index k and compare n pairs of *previously* calculated sets:

$(T_{ii}, T_{i+1,j}), (T_{i+2,j}), ..., (T_{i,j-1}, T_{jj})$

Graphically, taking again $|w| = 5$, if we are calculating the (final) entry T_{15} we proceed with the sequence of comparisons shown in Fig. 6.2 where we have the comparisons: (T_{11}, T_{25}), (T_{12}, T_{35}), (T_{13}, T_{45}), (T_{14}, T_{55})

6. Decision criterium: if the last entry of the matrix (the top-left element $T_{1,|w|}$) con-

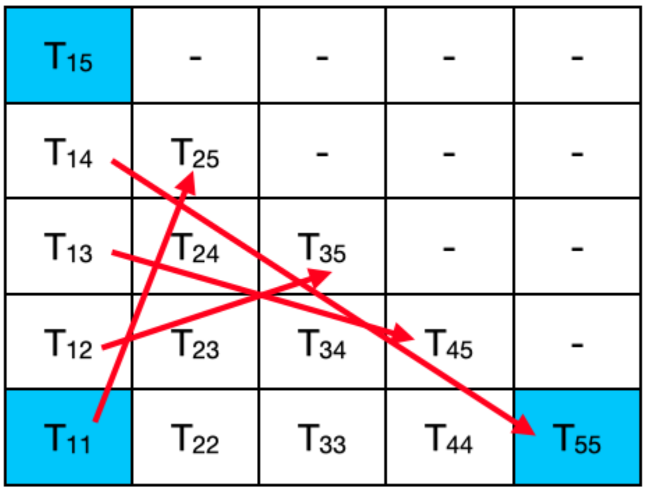


Figure 6.2: Sequence of comparisons during the determination of the matrix entry T_{15} in the CYK algorithm. Note the pattern of "rotation" of the arrows.

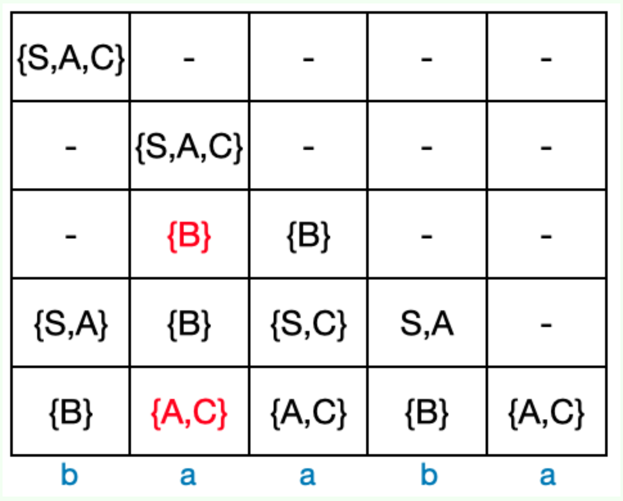
tains the start variable, then G accepts the string w, otherwise, it does not.

The working of the algorithm might be better understood with an **example** (taken from Hopcroft-Motwani-Ullman):

We consider the grammar G:

$$\begin{aligned}(1) : S &\to AB|BC\\ (2) : A &\to BA|a\\ (3) : B &\to CC|b\\ (4) : C &\to AB|a\end{aligned}$$

and the word $w = baaba$ with $|w| = 5$. The resulting CYK table is:



{S,A,C}	-	-	-	-
-	{S,A,C}	-	-	-
-	{B}	{B}	-	-
{S,A}	{B}	{S,C}	S,A	-
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

Let's see how the elements highlighted in red were obtained.

The element T_{12} belongs to the first row, i.e the "base-case" which is obtained without the help of previously calculated ones. It corresponds to the $i = |w| = 1$ word 'a'. The non-terminal 'a' can result only from the applicaiton of rules (2) and (4) whose heads are A and C, therefore, $T_{12} = \{A, C\}$
The other elements of the $i = 1$ row are computed analogously.
Let's now consider an element in a higher row: T_{24}. This corresponds to the word starting from the second place in w and ending on the fourth place: 'aab'. This word can be broken at the points $k = 2$ or $k = 3$ ('a-ab' or 'aa-b').
This means that we have to look in the entries :

$T_{22}T_{34} \cup T_{23}T_{44}$ = $\{A, C\}\{S, C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$

We found 5 strings, under which only CC corresponds to a body, which head is B (rule (3)), therefore $T_{23} = \{B\}$.

The algorithm runs over the length n of the string, the starting position (another factor n) and n possible splittings. Overall, the running time scales as $O(n^3)$.
Summarizing, CYK checks whether a word belongs to a context-free language by building all possible ways substrings can be generated (with CNF rules), starting from single non-terminals and combining them until the full string is covered. Since previous results are re-used (dynamical programming), the running time scales polinomially.
Since the matrix has $n(n-1)/2$ entries, the memory usage ("space") scales like $O(n^2)$.

6.10 Operations on the Context-Free Languages

Context-free languages can be subject to specific operations over them. The interesting operations are the ones with the *closure* property, i.e. are the ones producing a context-free language again.
Let's define first the set of all context-free languages with $\mathcal{C}$.

- **Union:** $L_1, L_2 \in \mathcal{C} \Rightarrow L_1 \cup L_2 \in \mathcal{C}$
  Proof idea: Given the two grammars G_1, G_2 generating L_1, L_2 respectively, construct a new grammar with a new start symbol S and rules $S \to S_1 | S_2$ where S_1, S_2 are the start symbols of G_1, G_2.

- **Concatenation:** $L_1, L_2 \in \mathcal{C} \Rightarrow L_1 \cdot L_2 \in \mathcal{C}$
  Proof idea: Construct a new grammar with rules like: $S \to S_1 S_2$

- **Kleene Star:** $L \in \mathcal{C} \Rightarrow L^* \in \mathcal{C}$
  Construct a new grammar with rules like: $S \to SS|\epsilon$ or similar rules allowing for repetition.

It is interesting to note that for the **intersection** $L_1 \cap L_2 \notin \mathcal{C}$ does not hold in general, as this couterexample shows:

- $L_1 = \{a^n b^n c^m : n, m \in \mathbb{N}_0\} \in \mathcal{C}$
- $L_2 = \{a^m b^n c^n : n, m \in \mathbb{N}_0\} \in \mathcal{C}$

L_1 and L_2 are context-free, since they are concatenations of context-free languages, but $L_1 \cap Ł_2 \notin \mathcal{C}$, since $L_1 \cap Ł_2 = \{a^n b^n c^n : n \in \mathbb{N}_0\}$ is not context free (see next section for a way to prove it and later on while discussing push-down automata) [2].

[2] A heuristic explanation is the following. $a^n b^n c^m$ are words with a constraint: the number of 'a's must be equal to the number of 'b's, while we can have an arbitrary number of 'c's. A pushdown automaton can push one symbol on the stack for each a, pop one symbol for each b, then read any number of 'c's without using the stack. Once the stack is empty, you no longer need memory and the 'c's are just "free": there is only one counting relation, and a single

6.11 The Pumping Lemma for Context-Free Languages

A version of the Pumping Lemma already stated for regular languages can be adapted to context-free languages. The lemma is a necessary (but not sufficient) condition for proving a language to be context-free.
In the regular languages cases, the strings of a language were divided in three parts: in the context-free case we will need five.
Intuitively, a context-free grammar has only finitely many nonterminals, so long strings must repeat some nonterminal along a derivation path. That repetition allows you to "pump" substrings up and down, producing infinitely many strings in the language.

Theorem 6.11.1: Pumping Lemma for Context-free Languages

If A is a CFL, there is a number $p \in \mathbb{N}_0$ (called 'pumping length') such that if s is any string in A with $|s| \geq p$, then s can be divided in five pieces

$$s = uvxyz$$

satisfying:

1. $\forall i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$,
3. $|vxy| \leq p$.

Proof. We prove here the theorem in an informal way for understanding the logic behind it. In Fig. 6.3 we consider a generic parse tree for a context-free grammar, able to generate a string s. If the string is 'long enough', at some point in the tree, a variable will be used at least twice (the variable is called A in the figure).
In particular:

- If the tree has at least $|V|+1$ levels ($|V|$ is the number of variables and there is a branching at each of them), then for the pidgeon principle, at least one variable is used more than one time.
- If the tree has maximum branching factor b and $|V|+1$ layers, then the final number of leaves is $p = b^{|V|+1}$.

stack is enough. On the contrary, $a^nb^nc^n$ is not context-free because we have two simultaneous constraints: the number of 'a's must be equal to the number of 'b's, which must be equal to the numeber of 'cs'. This is a problem, since a push-down automaton has only one stack.

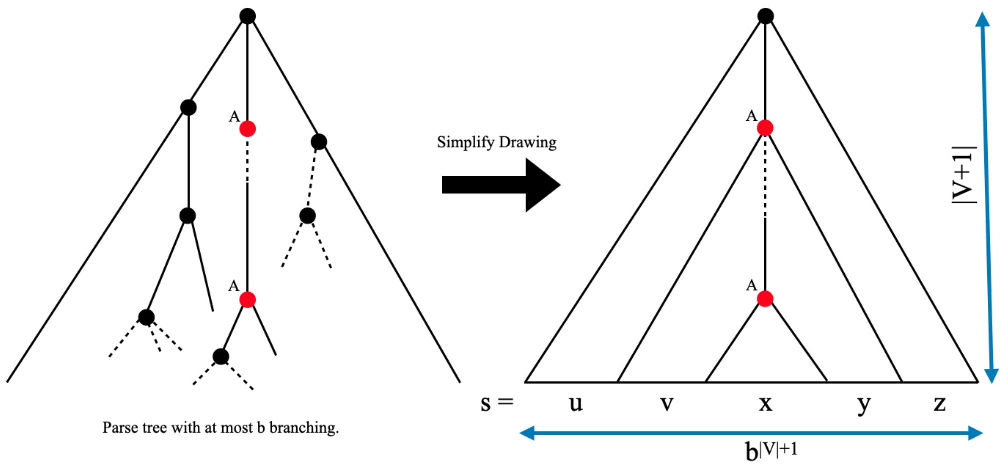


Figure 6.3: **(Left:)** A generic parse tree for a context-free grammar with branching at most equal to b generating a string $s = uvxyz$ (note that u,v,.. are sub-strings, not terminals only). **(Right:)** Same picture ion a simplified form. The minimal tree height and width are indicated.

- The number p is called the **pumping length** and a string length must be equal or larger than it in order to have at least one repetition of a variable in the tree.

In Fig. 6.3, we see that the final generated string is $s = uvxyz$: the first appearence of the variable A (the 'top' one) eventually (after a more or less complex path through the tree) generates the sub-string parts v and y. The second appearence of A is involved in the generation of the sub-string x.
In general, x is generated with a long chain after the first appearence of A: $A \Rightarrow ... \Rightarrow x$. This 'chain' is symbolically represented with $A \overset{*}{\Rightarrow} x$. We can then also say:

- Since the 'second' A implies $A \overset{*}{\Rightarrow} x$, the 'first' A leads to $A \overset{*}{\Rightarrow} vxy$.
- We can also say equivalently that $A \overset{*}{\Rightarrow} vAy$.
- This reasoning can be iterated $\forall i \geq 0$, leading to: $A \overset{*}{\Rightarrow} vvAyy...$ and therefore: $A \overset{*}{\Rightarrow} v^iAy^i$.

What we found is that the string can be 'pumped' to a sequence of strings all of which still belong to the context-free language.
Stated in other words, a CFG has only a finite number of variables ($|V < \infty|$), so when it generates a sufficiently long word (longer than the

pumping length), the corresponding parse tree must contain a repeated variable along some root-to-leaf path.
This repetition creates a subtree that can be duplicated (or removed) without breaking grammatical correctness. Therefore, sufficiently long words in a CFL must contain a "pumpable" region whose repetition yields new words that are still in the language.
The pumping lemma does **not** say that every string in a context-free language can be pumped or that failing to pump a string means the string is not in the language. It only says that if the language is context-free, then every sufficiently long string in the language admits at least one pumpable decomposition. **If pumping produces strings not in the language, then the language is not context-free** (the pumping lemma is a necessary condition for context-freeness, but not a sufficient one).
What we discussed, proves the first condition of the theorem.
The second condition states that $|vy| > 0$, which means that v and y cannot be both the empty string ϵ. Indeed, if they were both empty, we would obtain a final string which would be too short for allowing the repetition of the variable A: we constructed already the tree as the smallest possible allowing for the repetition.
The third condition states that $|vxy|$ is at most equal to the pumping length p........ ..... □

6.12 Example Application of the Pumping Lemma

As an example, let's consider again the language $L = \{a^n b^n c^n : n \in \mathbb{N}_0\}$ seen in Sec. 6.10 and try to prove that it is not context-free (it fails the Pumping-Lemma properties).
The Pumping Lemma is usually used in proofs by contradiction. So let's assume that L is a CFL and consider one of its strings $s = a^k b^k c^k$ for some $k \geq 0$. Clearly, $s \in L$ and $|s| \geq k$.

The Lemma assures us that s can be pumped, but we will see that there is no decomposition $s = uvxyz$ such that all three conditions of the theorem can be satisfied.

Considering condition two, $|vy| > 0$: one of the two substrings is non-empty.
We have two cases:

1. **v and y are made by just one character type.** In this case, v cannot contain both 'a's and 'b's or 'b's and 'c's. For example, uv^2xy^2z cannot contain an equal number of each non-terminal, violating condition 1. of the Lemma, leading to a contradiction.

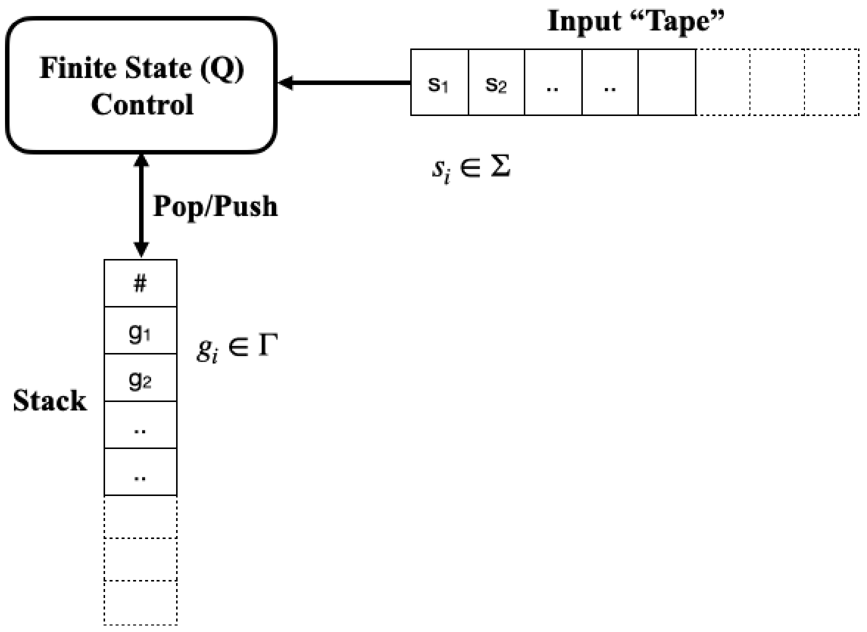


Figure 6.4: Schematics of a Push-Down Automaton.

2. v **and** y **are made by just more than one character type.** In this case, uv^2xy^2z can contain an equal number of the three symbols, but never in the correct order, leading again to a contradiction.

6.13 Push-Down Automata

Push-down automata (PDA) are an extension of the non-deterministic finite automata. The extension involves the presence of a **stack**, which is a (infinite) memory structure with first-in-last-out (FILO) structure. **We will see that PDAs have computational power equivalent to context-free grammars**.
In general, we can visualize the PDA as a machine which has a certain number of states (like the ones in FAs) and which accepts a tape as input where a "word" is written. At each step of the computation (each time the machine evaluates a "letter" on the tape), it can also consult a stack, reading/writing on it (see Fig. 6.4). More commonly, reading (removing) a symbol from the stack is an operation called **pop**, while introducing a new symbol in the stack is called **push**.
At each step of the computation, the PDA reads an input, and pops or pushes elements to/from the stack.

Why a stack makes a PDA more powerful than a FA can be understood considering a language not accepted by FAs, like $L = \{0^n1^n : n \in \mathbb{N}\}$. A PDA can accept L through the following computation (compare with Fig. 6.4):

1. Read the first symbol from the input (it will be a 0, since words are of the type 000...111.),
2. If the input is a ’0’, push it into the stack,
3. If the input is a ’1’, pop a ’0’ from the stack.
4. If the stack becomes empty as you read the last ’1’, accept.

The previous example shows that providing an automaton with a form of memory enlarges the family of accepted languages. A FA has a memory which coincides with its number of states and therefore cannot process words not fitting its structure.
A formal definition of a (non-deterministic) Push-Down Automaton is the following:

Non-Deterministic Push-Down Automaton

A non-deterministic push-down automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, F are finite sets and:

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : \ Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_o \in Q$ is the start state,
6. $F \subseteq Q$ is the set of accepted states.

with $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, and $\mathcal{P}(A)$ indicates the power set of A. Sometimes, for technical reasons, like for simplifying some proofs, a seventh element is added, namely $\# \in \Gamma$ marking the stack start symbol.

A key part of the definition is the transition function δ: it takes as input the current state, the input symbol, and the stack’s (top) content producing a new state and a new stack’s symbol which belong to $\mathcal{P}(Q \times \Gamma_\epsilon)$. This is an indication that the automaton is not deterministic: δ returns in general a **set of members of** $Q \times \Gamma_\epsilon$, not a single state and a single new symbol (compare with the discussion of non-deterministic FAs).

6.14 (Non-)Determinism in Pushdown Automata

For finite automata, nondeterminism is equivalent to determinism, because the machine has finite memory and all possible choices can be “tracked in parallel” inside a finite state.
A PDA instead has a finite control, but an infinite stack: if the automaton makes a "wrong" choice about when to push/pop which is discovered later in the computation, it is not possible to “go back”, since the stack history matters. Non-determinism helps discovering the structure in the input in advance and always following the “right” among the many which are explored.
Another way to see this is when we use a PDA for a parsing task. A deterministic PDA must decide, based only on the current state, input symbol, and stack top (and no possibility to “backtrack”). The problem is that many context-free languages require deciding which structure the input has before enough information is available. Non-determinism allows for "committing" to a certain structure while exploring all the possibilities in parallel.
A classic example for a language which is accepted by non-deterministic PDA but not from deterministic ones is:

$$L = \{ww^R : w \in \{a,b\}^*\} \quad .$$

L is the language containing all the words made by the a,b symbols which are even-length palindromes (R is the “reversing” operator: $(ab)^R = ba$). A deterministic PDA should understand somehow where the middle of the string is, but this position is not marked by anything (the length of the string is not known in advance). This means that if the automaton pushes/popps too early it will not accept the string.
A non-deterministic PDA can “guess” every position in parallel, including the right one.
Another example is $L = \{a^i b^j c^k : i = j | j = k\}$: in this case the automaton must decide if to compare ’a’ and ’b’ or ’b’ and ’c’ without any additional information.
A general conclusion is that deterministic context-free languages (D-CFL) are a strict subset of non-deterministic languages (ND-CFL):

$$\text{D-CFL} \subset \text{ND-CFL} \quad .$$

In general, non-determinism is not practically realizable but it can be simulated, although with exponential physical resurces. It models effectively search, choice, or incomplete information: this feature is not really about the “hardware” of the automaton, but it is about its expressiveness.

If D-CFLs are only a subset of a more general class, what are their specific properties?
D-CFLs have the following characteristics:

1. They are closed under the **complement**. Given a language $L \subset \Sigma^*$, the complement of L wih respect to Σ is $\bar{L} = \Sigma^* - L$, i.e. all the strings built with symbols in Σ which are not in the form specified by L. Regular languages are closed under complement, but in the context-free case only the deterministic ones are closed too.

2. are **unambiguous**.

3. parsing is efficient (O(n), in this case): this means also that a parser for such languages never needs to backtrack and only one stack is needed.

In particular, properties 2. and 3. show why many programming languages are based on D-CFLs (and the corresponsing grammar).

6.15 Push-Down Automata and Context-Free Languages

In Sec. 6.13 we anticipated that PDAs accept context-free languages. This concept is captured by the following key theorem:

Theorem 6.15.1: PDA/CFL Equivalence

A language L is context-free **if and only if** there is a push-down automaton accepting it.

Another equivalent formulation (reminiscent of search algorithms on trees) is:
Whatever a context-free grammar does with derivation trees, a PDA can simulate it using its stack.

We divide the proof in two parts, corresponding to the two implications.

Proof.

PART 1: L is a context free language ⇒ A PDA accepts it.

The idea of the he proof is to convert the context-free grammar G generating L into a PDA. We have to construct a PDA which accepts a word w if it was derived using the grammar G. The derivation is a sequence of applications of substitution rules dictated by G: which rule to apply at every step of the derivation is selected by the non-deterministic functioning of the PDA (the automaton always

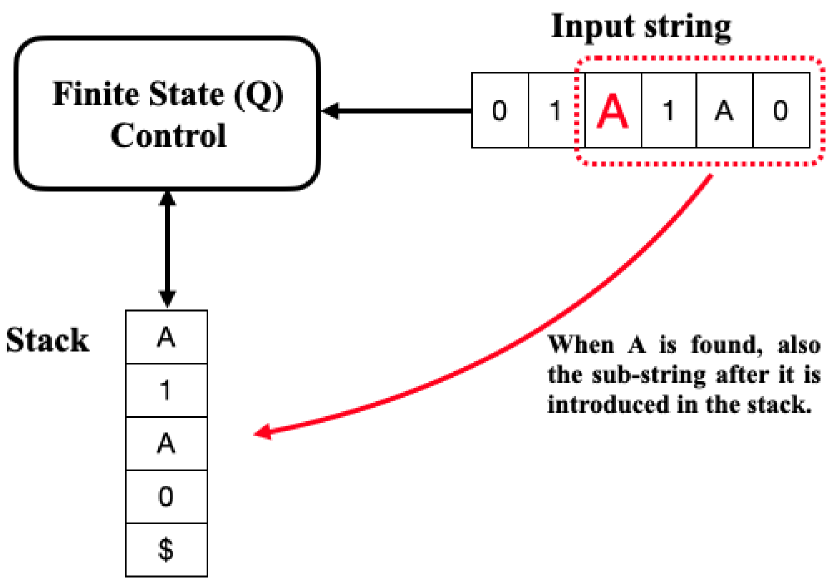

Input string
Finite State (Q) Control
0
1
A
1
A
0
Stack
A
1
A
0
$
When A is found, also the sub-string after it is introduced in the stack.

Figure 6.5: PDA functioning for PART 1 of Theorem 6.15.1.

can select the right intermediate rule for reaching the result).

A context-free grammar generates a string by repeatedly expanding nonterminals until only terminals remain. A PDA can simulate this procedure using a special procedure: The stack will store the "rest of the derivation still to be completed." This last idea is needed because we can only look at the top of the stack: if we store the whole intermediate string, then we have to look for the place where we have to perform the next substitution. What we do instead is to store in the stack only part of the intermediate string, i.e. only the symbols *starting* with the variable to substitute. This is basically the whole idea of the proof.
Initially, the PDA pushes the start symbol $ onto the stack and the input "reading head" is at the beginning of the word. At every step, the PDA looks at the top of the stack:

- If the top is a nonterminal (variable) A, the PDA nondeterministically chooses one of the grammar rules $A \to s$ and replaces the A in the stack with the string s.
- If the top is a terminal a, the PDA checks whether the next input symbol is also a.
  If yes, it consumes the input symbol and pops a from the stack. If not, the non-deterministic computation branch rejects.

Note that in the previous steps, we say: "replaces the A in the stack with the string *s*". This does not look possible, since we can push in the stack only one symbol at a time. This issue can be fixed introducing a transition function which can handle sub-strings with a "loop".

The PDA accepts if both the input is completely consumed and the stack is empty.
In other words: Grammars generate strings "top-down" by expanding symbols according to a tree-like structure. PDAs process strings left-to-right and the stack is used for keeping track of the unfinished work to do. The non-determinism here is essential.

PART 2: If a PDA accepts a language L $\Rightarrow$ L is context-free.

Here we are basically saying that the stack behaviour of a PDA can be encoded in context-free gramma rules.
A PDA accepts a string if there is a successful computation. For a PDA this means starting with an empty stack and end in an accepting configuration after having read the whole input. The computation is a "nested" process: symbols are pushed and popped and what happens between a push and a pop is an independent *sub-computation*. We will build a grammar whose nonterminals describe what a PDA can do between two stack configurations.

□

6.16 Programming Languages

We would like to see a concrete realization of a context-free language that we actually find in practice. Before doing that, we have to make a short digression about programming languages in general.

Many useful syntactic fragments of programming languages are context-free. HTML or XML with nested tags, arithmetic expressions with parentheses, block structures with matching braces: all of these are context-free patterns and are recognized by push-down automata (equivalent to context-free grammars). This is why parsers for programming languages are usually based on context-free grammars. This does not mean that full programming languages like C or Python are context-free languages in the formal sense. The key point is this:

A programming language is not just its syntax.

When we say “HTML is context-free” or “C has a context-free grammar”, we are talking about syntax only: which strings of characters are **well-formed programs**. A parser checks just this. Parsing just checks the grammatical correctness, not whether the code “makes sense”.

After this digression, we can turn to a concrete example and we will consider a rather simple language: `HTML` (HyperText Markup Language). Here we consider a sub-set of simple rules similar to `HTML` (the full rules would be much more complex):

Terminals:
`<tag>` `</tag>` (for various tag names)
TEXT (arbitrary text not containing < or >)

Rules:

$$\begin{aligned}
\text{Document} &\rightarrow \text{Element}\\
\text{Element} &\rightarrow \text{Text} \mid \text{Tag}\\
\text{Tag} &\rightarrow \langle\text{Name}\rangle\ \text{Content}\ \langle/\text{Name}\rangle\\
\text{Content} &\rightarrow \epsilon \mid \text{Element}\,\text{Content}\\
\text{Text} &\rightarrow \text{TEXT}\\
\text{Name} &\rightarrow \text{TAGNAME}
\end{aligned}$$

Here we considered only two terminals and a minimal set of rules. The most important rule is Tag $\rightarrow$ $\langle$Name$\rangle$ Content $\langle$/Name$\rangle$ which is able to give rise to structures similar to parenthesis matching in arithmetic expressions. In general the rules are able to generate:

- Arbitrary nesting of tags (recursion)
- Proper matching of opening and closing tags
- Mixing text and elements

for example code fragments like:

```
<div>
  Hello
  <p>world</p>
</div>
```

6.17 Context-Sensitive Languages

Context-Sensitive (Typ.1) languages are less common in applications and rather complex in their analysis. Here we just briefly introduce them. We remind that “context” refers to the surroundings of a rule and a context-sensitive language can pay attention to context. For example, rules like $abS \to aRb$ substitute strings where the rule S is preceded by the terminal string ’ab’. Such rules are not allowed in context-free languages. The computational model for context-sensitive languages is the **Linear-Bounded Automaton (LBA)** which is a non-deterministic Turing machine with tape restricted to the size of the input (or a linear function of it). The Turing machine will be defined in the next chapter, but here we anticipate that it is a generalization of the push-down automaton, where instead of a stack we have an infinite ’tape’ where we can move in both directions over it and the tape itself is potentially infinite. In the case of LBAs, the “memory” is a finite tape of length $O(n)$ if the input is of length n. Here the formal definition:

Linear-Bounded Automaton

A Linear-Bounded Automaton (LBA) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\textbf{accept}}, q_{\textbf{reject}})$$

where:

- Q **is a finite set of states,**
- Σ **is a finite input alphabet (without the blank symbol** $\sqcup$**),**
- $\Gamma \supseteq \Sigma \cup \{\sqcup\}$ **is the tape alphabet,**
- $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$ **is the transition function (for deterministic LBA,** δ **is a function),**
- $q_0 \in Q$ **is the start state,**
- $q_{\textbf{accept}} \in Q$ **is the accept state,**
- $q_{\textbf{reject}} \in Q$ **is the reject state,** $q_{\textbf{reject}} \neq q_{\textbf{accept}}$**.**

Linear-bounded restriction: Let the input $w \in \Sigma^*$ **have length** n**, and let the tape initially contain** w**. During computation, the head can only read and write within a tape of length at most** $c \cdot n + d$ **cells, for some fixed constants** $c \geq 1$ **and** $d \geq 0$**.**

An input $w \in \Sigma^*$ is *accepted* if starting from the initial configuration $(q_0, w, 1)$, the LBA eventually enters q_{accept} while staying within the bounded tape. Otherwise, it is rejected. Establishing if a string belongs to a context-sensitive language is solvable in polynomial space (`PSPACE-complete`) and therefore it can be quite inefficient.
Given their complexity, these languages have limited applications. For example, they can model certain structures found in natural languages but they are largely avoided in programming languages. Theoretically, they provide a natural bridge from context-free languages to the most general languages of Typ.0 which will be described in the next chapter.

7 Turing Machines and the Church-Turing Thesis

We introduce now a model of computation which is equivalent to a general-purpose computer and can accept formal languages without restrictions. In fact, it realizes the Typ.0 languages in the Chomsky hierarchy. The model is called **Turing Machine** from the name of its inventor, Alan Turing (1912-1954). Actually another model (the λ-calculus of A.Church) was proposed shortly before, but the Turing Machine (1936) is considered more intuitive and easy to understand and also it is surprisingly similar to how a real computer works: this is remarkable since at the time it was conceived, computers in their modern form were still not existing (Turing himself was involved in one of the first attempt to build one).

7.1 The Turing Machine

The Turing machine is an abstract model of computation which looks computer-like, while other (provably equivalent) models look more programming-like. A Turing machine must have something more with respect to -for example- push-down automata. For example, the "memory" must be less restricted and therefore we will consider an infinite "tape" where the machine can read a symbol. The machine will be able to move its reading head sequentially in both directions (thus it can do more than a pop-push operation), plus it will have a set of "states" in which it can be found after reading/writing on the tape (see Fig. 7.1).

Formally, the Turing machine can be defined as follows:

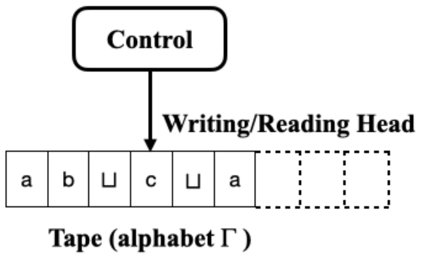


Figure 7.1: Schematic representation of a Turing Machine.

Turing Machine

A Turing Machine T is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

1. Q is a finite set of states
2. Σ is a finite set of symbols without the 'blank' symbol $\sqcup$: the input alphabet
3. Γ is the 'tape' alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state
6. $q_{\text{accept}} \in Q$ is the accept state
7. $q_{\text{reject}} \in Q$ is the reject state and $q_{\text{accept}} \neq q_{\text{reject}}$

7.2 Turing Machine Computation

Before the machine starts, the tape contains exactly the input string over Σ followed by blanks. Basically Σ is the problem specification alphabet and belongs to the language being decided. The alphabet Γ contains also Σ but pertains to the machine's internal working. In very simple machines, it can be that $\Gamma = \Sigma \cup \{\sqcup\}$ but in general more symbols are required for the computation on the tape. For an analogy with real computers, we can think of Σ as the alphabet for the files format, while Γ contains the symbols used in the RAM memory while the computer executes an algorithm.
This is how a Turing machine T performs a computation:

1. T is in the start state q_0

2. T receives a string as input: $w = w_1 w_2 ... w_n \in \Sigma^*$ on the leftmost n places (or 'squares') of the tape, while the other places are filled with blank characters $\sqcup$.

3. The reading/writing head is initally placed on the left-most square.

4. Since Σ does not contain the blank symbol, the first blank marks the end of the input string.

5. At the start of the computation, what happens is governed by the transition function δ.

6. The transition function has as input the current machine's state and the symbol read from the tape.

7. The transition function outputs a new state, a new symbol (which overwrites the one read as input), and issues the command to move the head to the left or to the right. If the head must move left but it is already at the beginning of the tape, it stays where it is.

8. At the end of the computation:
   - T reaches one of the two *halting configuations*: q_{accept} or $q_{\text{reject}} \in Q$, or
   - T never stops (enters a 'loop' state).

We say that the Turing machine **accepts** if it reaches q_{accept} after traversing a finite sequence of states.
The set of strings accepted by T is the **language of T** or the **language recognized by T**.
We have therefor the definition:

Turing-recognizable languages

A language is **Turing-recognizable** if there is a Turing machine recognizing it. Such languages are also called **recursively enumerable languages**.
Recalling the classification of Ch. 3, these are the Typ.0 languages in the Chomsky hierarchy.

As we have seen, a Turing machine can accept, reject, or never stop (looping forever). Considering only machines which eventually stop (either in the accept or reject state), we call them **deciders**. A decider which also recognizes a language then **decides** that language. This leads to the definition:

Turing-decidable languages

A language is **Turing-decidable** if there is a Turing machine able to decide it.

In order to better understand the computational flow of a Turing machine T, let's see how T works on a simple language:

$$L = \{w \in \{0,1\}^* | w \text{ contains at least one } 1\} \quad .$$

T must accept strings like '00100', '100' and so on, while rejecting ϵ, '00', '0000', ...
The idea of such simple machine is to accept whenever you encounter a '1' character.
The tape alphabet is $\Gamma = \{\sqcup, 0, 1\}$ and the states $Q = \{q_0, q_{\text{accept}}, q_{\text{reject}}\}$.
The transition function can be given in tabular form:

Current state	Read	Write	Move	Next state
q_0	0	0	R	q_0
q_0	1	1	R	q_{accept}
q_0	$\sqcup$	$\sqcup$	R	q_{reject}

Or in a more compact, function definition style:

$$\begin{aligned}
\delta(q_0, 0) &= (q_0, 0, R) \\
\delta(q_0, 1) &= (q_{\text{accept}}, 1, R) \\
\delta(q_0, \sqcup) &= (q_{\text{reject}}, \sqcup, R)
\end{aligned}$$

This example is extremely simple and only right-moves are involved. A slightly more complex example is the one involving the language

$$L = \{0^n 1^n | n = 1 \text{ or } n = 2\} \quad .$$

Here the tape alphabet is larger than before $\Gamma = \{0, 1, X, Y, \sqcup\}$ and we need also more states: $Q = \{q_0, q_1, q_2, q_{\text{accept}}, q_{\text{reject}}\}$.
This time both, left and right moves are required, and the transition function is

Current state	Read	Write	Move	Next state
q_0	0	X	R	q_1
q_0	X	X	R	q_0
q_0	Y	Y	R	q_0
q_0	$\sqcup$	$\sqcup$	R	q_{accept}
q_1	0	0	R	q_1
q_1	X	X	R	q_1
q_1	Y	Y	R	q_1
q_1	1	Y	L	q_2
q_1	$\sqcup$	$\sqcup$	L	q_{reject}
q_2	0	0	L	q_2
q_2	X	X	R	q_0
q_2	Y	Y	L	q_2
q_2	$\sqcup$	$\sqcup$	R	q_0

Representing the head position as a bar over the corresponding symbol and the current state before the string, we can follow the computation of the string '0011':

$$\begin{array}{ll}
q_0 & \overline{0}\,0\,1\,1\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_1 & X\,\overline{0}\,1\,1\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_2 & X\,\overline{0}\,Y\,1\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_0 & \overline{X}\,0\,Y\,1\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_1 & X\,X\,\overline{Y}\,1\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_2 & X\,X\,Y\,\overline{1}\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_0 & \overline{X}\,X\,Y\,Y\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_0 & X\,X\,Y\,\overline{Y}\ \sqcup\ \sqcup\ \sqcup\ \ldots \\
q_{\text{accept}} & X\,X\,Y\,\overline{Y}\ \sqcup\ \sqcup\ \sqcup\ \ldots
\end{array}$$

The idea (the algorithm) behind the definition of this machine is:

1. Start at the leftmost square of the input.
2. Find the *leftmost 0 that has not yet been marked, and mark it (replace it with the symbol X).*
3. *Move right to find the* rightmost 1 that has not yet been marked, and mark it (replace it with Y).
4. Move back left to the beginning of the tape to find the next unmarked 0.
5. Repeat steps 2–4 until all 0s and 1s are matched.
6. If all symbols have been successfully matched, *accept* the input; if a mismatch occurs, reject the input.

7.3 Multi-tape Turing Machine

Apparently, the single tape of a Turing machine might seem like a limitation to its computational power. We can define a Turing machine with a number k of tapes through the transition function

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

It is not difficult to convince ourselves, that having more than one tape does not help in increasing the computing power of the machine, or in other words, to compute more functions than the original single-tape machine. In fact, we have the following theorem:

Theorem 7.3.1: Multitape Turing Machine Equivalence

Every Multitape Turing Machine has one equivalent single-tape Turing Machine.

The idea of the proof is to construct a new Turing machine with only one tape, but able to simulate a k-tape machine. To this aim, we can just include in the tape alphabet a symbol (e.g. '#') for separating the different tapes and new symbols identical to the normal ones s, but with an additional mark like $\bar{s}$ for remembering where the head is inside each portion of the tape:

$$\text{Sigle tape} = ab\bar{c}adf \,\#\, abdc\bar{a}fg \,\#\, \bar{c}adfb \,\#\, ... \quad .$$

The computation of such a Turing machine will be much more contrived and with a more complex transition function, but it will be fully equivalent to the k-tapes machine.
We will discuss more in Sec. 7.7 about equivalent models of computation but it turns out that whatever modification you can try to do to a Turing machine for enhancing its computational power, this will remain the same of the simplest, original version defined by A. Turing.
Still, a Turing machine with more tapes might seem "faster", even though it does not enlarge the class of accepted languages. This efficiency improvement is formalized with the following theorem:

Theorem 7.3.2: Multi-Tape Turing Machine Time Complexity

A k-tape Turing Machine running in time $T(n)$, where n is the length of the input string can be simulated by a single-tape Turing machine in time $O([T(n)]^2)$.

The penalty of having only one tape results in a quadratic slow-down with respect to a multi-tape machine. This happens because the $k = 1$ machine encodes the content of the $k > 1$ tapes sequentially, separating them with a marker. The $k = 1$ machine must for each step of the $k > 1$ machine:

1. Locate the head markers scanning the whole tape: this will take $O(k \cdot T(n)) \sim O(T(n))$ time,
2. Determine the transition (constant time),
3. Scan again to update the symbols and move the heads left/right: $O(T(n))$.

The procedure has an overall time cost of $O(T(n) \cdot O(T(n))$. Summarizing intuitively, while the $k > 1$ machine acts on many tapes locally, the $k = 1$ machine has to scan a $O(T(n))$-long tape in addition to the $O(T(n))$ time for the computation.
With clever encodings this bound can be tightened. For example, the Hennie-Stearns method (1966) can achieve $O(T(n \cdot \log T(n)))$ leveraging on a tree-like data structure on the single tape. While it is impossible to reach $O(T(n))$ for a $k = 1$ machine, there is no proof that the Hennie-Stearns is optimal and cannot be improved. Still, it is the best bound known.

7.4 Non-Deterministic Turing Machine

It is possible to design a non-deterministic version of the Turing machine (NT), in the same way we defined for example non-deterministic finite automata. A NT will be able at each step to explore all the possible moves and pick the right one. This means that the transition function becomes:

$$\delta : Q \times \Gamma \to \mathcal{P}\left(Q \times \Gamma \times \{L, R\}\right)$$

The power set on the right-hand side represents a (potentially huge) tree of possible computational paths of the machine, which always chooses the right branching leading to an accept state.
A relevant characterization theorem about NTs is the following:

Theorem 7.4.1: Non-deterministic Turing Machine Equivalence

Every non-deterministic Turing machine has an equivalent deterministic Turing machine.

The previous theorem states (again) that even the non-deterministic version of a Turing machine is not more powerful than the original Turing machine: the set of computable functions (or accepted languages) is the same for both. A relevant difference from the

point of view of **computational complexity theory** is that if a deterministic Turing machine simulates a non-deterministic one (as it can), it can incur in at most **exponential slowdown**.
Why at most exponential and not something even more severe? This is because a non-deterministic Turing machine defines a computation tree: At each step, there are at most b non-deterministic choices so if for each choice the machine takes time $T(n)$ (n is the length of the input) the depth of the tree is at most $b^{T(n)}$.
Let's see now the proof of the last theorem:

Proof. The idea behind the proof is to let a deterministic Turing machine D try all the paths of the non-deterministic one N until the accept state is reached. Like in the latter observations about the exponential slow-down, we visualize the computational process on a tree with the root as starting configuration of the machine.
We will explore the tree with a breadth-first search method, since the depth-first might "go down" forever in the case the current branch does not contain an accepting state.
For this proof, we will leverage on the result of Theorem 7.3.2 stating that multi-tape Turing machines are equivalent to single-tape ones. Here we consider a 3-tape machine where:

1. **Tape 1:** Contains the input string and it is never (over)written.
2. **Tape 2:** Contains a copy of the tape of N in the current computational branch.
3. **Tape 3:** Records the location of D on the N computational tree. It effectively contains a "memory address".

Here is how we write on Tape 3:

- Every node in the tree has at least b children (b is the largest number of choices possible)
- Every node has an address expressed as a string over the alphabet $\Sigma_b = \{1, 2, 3, ..., b\}$.
- For example, we reach the address '234' taking the second branch from the root, then the third, and then the fourth.
- Thus Tape 3 contains a string of numbers which uniquely identifies a node on the N computational tree. The address might also be invalid (not corresponsing to an existing node of N).

- The empty string identifies the root node.

Having this addressing system, we show how D can simulate N:

1. At the start, Tape 1 contains the input string and the other two tapes are empty. We need Tape 1 remembering the input because every time we restart a new branch we have to re-initialize the computation.

2. Copy Tape 1 on Tape 2.

3. This is the key step where D simulates N. Now D consults Tape 3 to determine which choice to make among all the ones followed in parallel by N. If Tape 3 is empty or the address is invalid, go to point 4. If an accepting configuration is reached, accept the input. Tape 2 effectively simulates one path of N.

4. Replace the string on Tape 3 with the next string according to the lexicographical order. Simulate the next branch of N's computation by going on step 2. This step basically enumerates all the possible choices (even the invalid ones).

The non-deterministic machine somehow takes always the right path to reach the accept state among all the possible ones. The deterministic one does not have this feature and in order to simulate this "magic" it must try all the possible paths, one by one. In this sense the non-deterministic machine does not try all the paths in parallel, but it makes always the right choices! □

7.5 Equivalent Models of Computation

The Turing Machine is not the only existing model of computation. While we considered models which are weaker than a Turing machine (finite automata, push-down automata, linear-bounded automata), do exist models which are computationally more powerful? The interesting answer is that it looks like we cannot invent computational models which are able to calculate functions (or accept languages) belonging to supersets of the one of a Turing machine. Researches just came up with different models which can be proved to be equivalent to a Turing machine: these observations lead to the **Church-Turing Thesis**.
In the following, we will consider some of these equivalent models. While Turing machines emphasize memory as a tape and local rewriting, we will study models emphasizing other aspects of the computation while trying to understand what are the core properties allowing the maximum computational power.

7.5.1 The Multi-Stack Automaton and Hypercomputation

Before the Turing machine, we considered push-down automata equipped with a memory organized as a stack. We can ask if more stacks might help in improving the class of accepted languages (or calculable functions). In turns out that with two stacks, we can simulate the whole tape of a Turing machine (the part to the left and to the right of the head). Therefore, a 2-stack push-down automaton is equivalent to a Turing machine.
The next question is if more than 2 stacks can help in improving the computational power but the answer is negative.
What limits context-free languages (push-down automata) is not syntax, but memory topology. You can have even infinitely many stacks but only finitely many are ever used, you still get Turing machines equivalence.
What would we really need to go beyond Turing calculation power are infinitely many stacks and the ability to unbounded parallel access to all of them, but this would not be a reasonable model of computation anymore. For having *hypercomputation* (go beyond the capabilities of a Turing machine), we can think of a machine with infinite memory which can be accessed randomly and an infinite amount of times in parallel. Such infinitely-parallel and infinite-memory machine could solve the halting problem (see later) or prove arbitrary logic formulas. Such machine can write infinite amounts of memory in one clock-tick so it can handle real numbers with infinite precision. Such a hypercomputer cout also calculate all the values of the BB-function, since it can Simulate all n-state Turing machines at once, in parallel, for unbounded lengths of time!

7.5.2 The Register Machine

The **Register Machine** is a simple model of computation which closely resembles **Assembler** programming and the functioning of a microprocessor. One of the most basic definitions is the following:

Register Machine

A **Register Machine** is a 2-tuple $RM = \{R, L\}$ where

- R is a finite set of **registers** $R_1, R_2, R_3, \ldots$. Every register can be tought as a memory location able to record one single natural number (they are sets with only one element: $0, 1, 2, \ldots$).
- L is a finite set of **instructions**.

While R can beseen as the "hardware" of the machine, L is the "software".
An example of possible instruction set is:

1. `INC(i,L)`: Increment by 1 the register R_i and go to instruction L.
2. `DEC(i,L`$_1$`,L`$_2$`)`: If the content of $R_i > 0$, decrement by 1 and go to instruction L_1, otherwise go to L_2.
3. `HALT`: stop the computation.

This machine can reach a halting state or continue to calculate forever. In particular, it is able to calculate all the *partial functions* $f : \mathbb{N}^k \to \mathbb{N}$.
How such a machine can work can be demonstrated with an example where we will try to add two numbers. What we want to realize is: $R_0 = R_1 + R_2$. This means that we have three registers: two contain the numbers to add and the third (R_0) is empty and will be used for storing the result.
This is the program (set of instructions L) for this task:

```
L1: DEC(1, L2, L3)    // while R1 > 0
L2: INC(0, L1)        //   R0++

L3: DEC(2, L4, L5)    // while R2 > 0
L4: INC(0, L3)        //   R0++

L5: HALT
```

Note that the instruction L_1, L_2 and L_3, L_4 are equivalent to **while loops** which keep decrementing the starting registers and increment the "sum" register R_0.
A key result about these machines is due to Minsky[1]

[1]Marvin Lee Minsky (1927-2016)

Theorem 7.5.1: Minsky's Theorem

A register machine with **two** registers and the instructions defined above is Turing complete. This means that a 2-register machine has the same computational power of a Turing machine.

In the following, we just sketch the proof in an intuitive way.
The key observation is that what makes a Turing machine powerful is not the tape itself, but the fact that it has *unbounded memory* (the tape can be infinitely long) and a finite control. A tape is just a very structured way of storing information. If you can store arbitrarily large integers and manipulate them, you can encode that same information in numbers instead of on a tape.

We can construct an encoding like this:

- Consider a Turing machine whose tape alphabet has finitely many symbols. Choose a numbering system with base k large enough to represent each symbol as a digit.
- Split the tape at the head position. Everything to the left of the head is encoded as one natural number, written in base k, with the symbol closest to the head in the least significant position.
- Everything to the right of the head, including the symbol currently under the head, is encoded as another natural number, again in base k, with the head symbol in the least significant position. So at any moment, the entire infinite tape is represented by just two integers.

With this encoding, head movement and symbol rewriting become just arithmetic operations. Moving the head to the right corresponds to dividing one number by k and multiplying the other by k, while moving the head to the left is the opposite.
Reading/writing symbols corresponds to looking at or modifying the least significant digit of one of the numbers. All of these operations can be built only with increments, decrements, and zero tests.

In summary, the key insight is that a Turing machine configuration is a finite object and therefore can be encoded as numbers. Two unbounded registers are enough to hold and manipulate such encodings. The defined (very small) instruction set is enough to implement the required arithmetic and therefore, 2-register machines can simulate Turing machines step by step. Another interesting and somewhat surprising conclusion is that computation does not fundamentally require "tapes", "symbols", or even a large "memory". Once we have (unbounded) natural numbers and conditional control flow, Turing computational power is already realizable and hidden inside pure arithmetic [2].

[2]Knowing Gödel's incompleteness theorem, probably many connections to it are starting

7.5.3 The WHILE Program

Thinking at the Register Machine, everything is done with a loop and a condition check (`DEC(i,`L_1`,`L_2`)`). This observation lead us to conclude that we can build an extremely simple programming language with only two instructions (`while` and `if`) for simulating completely a Turing machine.
Indeed such computations model exists and it is commonly called the **WHILE Program**:

WHILE Program

A WHILE program is defined over a fixed, finite set of variables $x_1, x_2, ..., x_n$ and $\forall i, x_i \in \mathbb{N}$.
A **state** is a function (assignment): $\sigma : \{x_1, x_2, ..., x_n\} \rightarrow \mathbb{N}$.
We **inductively** define now a **syntax** for programming this machine:

- **Arithmetic Expressions:** $a ::= x|0|a + 1|a - 1$ with subtractions truncated at zero.
- **Boolean Expressions:** b::=a=0
- **Commands:** `skip`|x:=a| C_1, C_2 |`if` b `then` C_1 `else` C_2 | `while` b `do` C.

It is not difficult to realize the similarities with the Register Machine. Note that a `for`-loop would weaken the computaitonal power of the machine and the `while` plus a condition is really needed. Placing fixed bounds to the loops would not realize the full power of a Turing machine [3]. The "sum program" we have seen for the Register machine can also be implemented with the WHILE Program syntax for $x0 = x1 + x2$:

```
while not(x1=0) do
    x0 := x0 + 1;
    x1 := x1 - 1
end;
```

to appear: encoding in numbers and the need to express arithmetic over the infinite natural numbers.

[3]WHILE programs (and Register machines and Turing machines) capture **partial computability**, which means the following. A WHILE program, given some input, may or may not terminate. If it terminates, it produces an output; if it does not, the function is undefined on that specific input.
The mathematical object corresponding to such a program is therefore a **partial function** from natural numbers to natural numbers.
Instead, a function is **total** if it is defined for *every input*, i.e. the computation always halts. If you restrict yourself to programs that are guaranteed to terminate, you get a strictly smaller class of functions: the **total functions**

```
while not(x2=0) do
    x0 := x0 + 1;
    x2 := x2 - 1
end
```

7.5.4 λ-Calculus

This computational model was proposed by Church a bit before Turing constructed his machine. While a Turing machine computes by moving in space, λ-calculus computes by "collapsing" structures. λ-calculus shows (again!) that computation doesn't need "machines" or "memory" but just **functions and substitutions** can realize Turing-equivalent computational power. Interestingly, **Fuctional Programming** is based on λ-calculus: one example is the `Haskell` programming language.
Here we give just a brief introduction in order to get the idea behind it. This model defines functions as

$$\lambda x.t \quad ,$$

where λx defines the variable, while the body of the function is given by the expression after the "dot". For example, in common function formalism $\lambda x.x + 1$ means $f(x) = x + 1$ but note that in the λ notation we did not give a name to the function (like "f" in the usual notation).
Application of functions is specified as an additional variable after function definition. For example for $f(x) = x + 1$ the application $f(3) = 3 + 1$ is defined as $(\lambda x.x + 1)3$.
We can also specify functions of more variables with expressions like

$$\lambda x.\lambda y.x + y \quad ,$$

which is equivalent to something like $f(x, y) = x + y$. The λ-calculus expression involves a recursive application of functions and it is more natural to translate the latter formula with $f(x) = (g(y) = x + y)$: this way of writing functions is called **currying**.
There is also a way for specifying all the natural numbers through the **Church Numerals**:

$$\begin{aligned}
&0 = \lambda f.\, \lambda x.\, x \\
&1 = \lambda f.\, \lambda x.\, f(x) \\
&2 = \lambda f.\, \lambda x.\, f(f(x)) \\
&3 = \lambda f.\, \lambda x.\, f(f(f(x))) \\
&\ldots\ldots \\
&n = \lambda f.\, \lambda x.\, \underbrace{f(f(\ldots f}_{n \text{ times}}(x)\ldots))
\end{aligned}$$

If f is the $f : y \to y + 1$ function when $x = 0$, then the latter expressions give exactly the natural numbers, otherwise they are just expressions saying "the number of times a function is applied".
Now that we have all the numbers and functions (with repeated applications) we can define also operations like addition:

$$\text{ADD} = \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x)$$

In order to see how all this can mimic the previous machines, we need boolean calculus for implementing conditions:

$$\text{TRUE} = \lambda t.\, \lambda f.\, t \quad ,$$

$$\text{FALSE} = \lambda t.\, \lambda f.\, f \quad ,$$

$$\text{IF} = \lambda b.\, \lambda x.\, \lambda y.\, b\, x\, y \quad .$$

For example:

$$\text{IF TRUE } a\, b = a \quad ,$$
$$\text{IF FALSE } a\, b = b \quad .$$

For an analog of the `while` we have the "Y-combinator":

$$Y = \lambda g.\, (\lambda x.\, g(x\, x))\, (\lambda x.\, g(x\, x)) \quad .$$

The outer λ indicates that Y takes g as input: this is the function we want to make recursive.
The inner part $(\lambda x.g(xx))(\lambda x.g(xx))$, calling $H = (\lambda x.g(xx))$ can be rewritten as HH.
H is a function that applies g to (x x). With HH, the x in H becomes H itself.
This allows g to indirectly call itself, because inside g we can use the argument (which is H) as the recursive call.
The combinaroe, in conjunction with a condition becomes:

$$\text{WHILE} = Y\, (\lambda w.\, \lambda x.\, (\text{IF}\, (\text{condition}\, x)(\lambda y.\, w(f\, x))x))$$

f is the function that updates the state, x is the initial state, and w is the recursive function provided by the Y-combinator. The combinator gives the function w a way to call herself [4].

[4] A typical recursively-defined function is the factorial. In terms of λ-calculus:

$$\text{FACT} = \lambda f.\lambda n.\text{IF}(n = 0)1(n \cdot f(n + 1))$$

Intuitively, if we can define all the natural numbers, `while` and `if`, this means that λ-calculus should be able to simulate a WHILE-Program model, which is equivalent to a Turing machine. So also λ-calculus should be again equivalent in terms of computational power to a Turing machine.

7.6 Functional Programming and λ-calculus

A functional programming language is a programming language where computation is done primarily by applying and composing functions, rather than changing state or using mutable variables. It emphasizes pure functions, higher-order functions (functions as values), and often supports recursion and immutable data.
Advantages if these languages are predictable codes (pure functions always produce the same output for the same input), easier debugging, modularity and reusability, concurrency-friendliness, and they are closer to mathematical reasoning [5].
For example, `Haskell` is a purely functional programming language and everything in it is a function. In this language, the recursive factorial function looks like

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

`Haskell` implements the core ideas ofλ-calculus. For example, the expression $\lambda x.x + 1$ becomes:

```
\x -> x+1
```

and the application $(\lambda x.x + 1)(3)$ is

```
(\x -> x+1)3
```

The addition $\lambda x.\lambda y.x + y$ becomes

```
add :: Int -> Int -> Int
add x y = x + y
```

or in λ-notation

```
add = \x -> (\y -> x + y)
```

where you explicitly see the *currying* of a function of two variables.

[5]See e.g. “Why Functional Programming Matters” from John Hughes (University of Glasgow)

7.7 The Church-Turing Thesis

Many computational models were proposed (see previous section) and all of them resulted equivalent to the power of a Turing Machine (but never more powerful). This lead to the formulation of a conjecture, known as the **Church-Turing thesis**, which is one of the main philosophical points of computability theory (maybe THE main point!). There are many ways to formulate it and here we propose one:

Church-Turing Thesis

Any function regarded as effectively computable, can be computed by a Turing Machine. "Effectively" means "computable by a mechanical, step-by-step" procedure.

Stated in a different way, every model of computation will be able to compute the same set of functions of a Turing machine (or a proper subset of it). The Turing machine is the most powerful "computer" we can think of.
One can also extremize the thesis and promoting it to a definition of effective computation.
This is still a *conjecture*, not a theorem, but so far it has proven to be extremely successful.

7.8 The Extended Church-Turing Thesis

The Church-Turing thesis seems having to do with the ideal world of mathematics and algorithms, but properly extended could be regarded as a statement about the physical world. This is the content of the **Extended Church-Turing thesis** (which is not due to Church or Turing):

The Extended Church-Turing Thesis

Any computation that can be *efficiently* performed by a physically realizable system can be efficiently simulated by a probabilistic Turing machine (A. Church, 1936).

"Efficiently" in this context means polynomial time. In the context of computational complexity theory, we can say: `BPP` (classical randomized polynomial time) is equivalent to "All physically efficient computation".

Another characterization of the thesis is: Any physically realizable computational system can be *efficiently* simulated by a (probabilistic) Turing machine.
So, Turing machines can simulate any "reasonable" computational model and if the original system runs in polynomial time, the simulation only incurs polynomial overhead.

For fully understanding the extended thesis, some notions of computational complexity theory are needed (the knowledge of the complexity classes like `P, NP, BPP`, ...) and which are not treated here [6]. The extended thesis is not just about computability: it can be also interpreted in relation to the computational power of the physical universe. No physical process can efficiently solve problems that are *intractable* for classical computers (for example, `NP-complete` problems in polynomial time. Computers at the end are physical systems, subject to the laws of physics and it seems like the Extended Thesis tells us an additional characterization of them: if a physical theory allows more computational power than a Turing machine (**hypercomputation**, see also Sec. 7.5.1), then something must be wrong in it.

Quantum computing challenges (and likely refutes) the Extended Church–Turing thesis, since we have indications that $BQP \subseteq BPP$ might be false (for example from Shor's Algorithm). Quantum computing does not violate the original Church–Turing thesis (a quantum computer can be simulated with a Turing machine) but only the extended version: uncomputable functions remain uncomputable also for a quantum computer.

If we believe that quantum computers can do more than solving only `BPP` problems, we can upgrade the extended thesis to a quantum version:

[6] Many excellent texts on this topic are available, e.g. a modern exposition can be found in: S. Arora, B. Barak, *Computational Complexity*, Cambridge.

Quantum Church-Turing Thesis

Any computation that can be performed efficiently by a physically realizable system can be efficiently simulated by a quantum Turing machine.

At this point, we need a definition for a Quantum Turing Machine: this will be discussed in another chapter.

8 Decidability

8.1 Introduction

Decidability is a key concept in computation that formalizes the question of what can be solved by an algorithm *in principle*, independently of efficiency or practical limitations of a physical machine. At its core, decidability defines which yes/no problems (decision problems) are mechanically (algorithmically) solvable and which not.

In this context, problems are represented as **languages** (sets of finite strings over an alphabet). A decision problem becomes the question of whether a given input string belongs to a particular language. This shift from informal "problems" to formal languages is crucial because it allows reasoning about computation using mathematical objects. Algorithms are modeled by abstract machines like Turing machines, which capture the intuitive notion of a step-by-step procedure.

A language is said to be decidable if there exists a Turing machine that, given any input string, always halts and correctly decides whether the string is in the language. Such a machine is called a decider (see also Sec. 7.2). Decidability therefore requires two properties: correctness and termination on all inputs. This distinguishes decidable problems from those that are merely computable but an algorithm may run forever on some inputs.

The study of decidability reveals that there are inherent limits to computation. Some well-defined problems cannot be decided by any algorithm, no matter how much time or memory is available. These undecidable problems are not artifacts nut are provably unsolvable within the framework of mechanical computation itself. Examples include the **Halting Problem** but also the calculation of functions, like the **Busy Beaver Function**.

Turing machines play a dual role in this theory. They are both the model of computation used to define algorithms and the objects about which algorithms reason. In the following we will basically leverage on the fact that we can encode a Turing machine into a number and feed it in another Turing machine.

Decidability is also closely connected to **reductions**, which provide a way to compare the difficulty of decision problems. By transforming instances of one problem into instances of another in a computable way, we can show that undecidability propagates: if an undecidable language reduces to another language, then the second language must also be undecidable.

Decidability describes the ultimate expressive power of algorithms. It tells us which questions are, in principle, answerable by computation and which lie beyond algorithmic reach. In this sense, the theory of decidability is not about computers as physical devices but about the fundamental nature of formal reasoning, computation, and the limits of mechanized problem solving.

8.2 Connection to Gödel's Incompleteness

Gd̈el's incompleteness theorems and the theory of decidability are connected because they both discover fundamental limits on formal, mechanical reasoning even though from different angles. Gd̈el worked within axiomatic systems for mathematics, while decidability theory works in the setting of algorithms and computation. The connection between them is the idea that proofs themselves can be treated as finite, manipulable objects, like programs or strings.

Gd̈el's first incompleteness theorem (1931) shows that any sufficiently expressive, consistent formal system for arithmetic contains true statements that cannot be proven within the system. At the time when the theorem was presented, it represented the end of Hilbert's dream of assiomatizing all the mathematics starting from integer numbers. David Hilbert formulated his program in the early 1920s proposing to formalize mathematics in precise axiomatic systems and that these systems should satisfy three key properties: **consistency** (no contradictions can be derived), **completeness** (every meaningful mathematical statement can be proved or disproved), and **decidability** (there exists a mechanical procedure to determine whether any given statement is provable).

Gödel's results implied that if there were a mechanical procedure that could decide the truth or falsity of every arithmetic statement, then every true statement would in principle be provable, contradicting incompleteness! In this way, incompleteness implies that there can be no algorithm that decides all truths of arithmetic. Decidability theory makes this intuition precise by replacing "truth" and "provability" with languages and algorithms.

Incompleteness and undecidability are two expressions of the same underlying phenomenon: the limits of formal systems that are powerful enough to talk about arithmetic and computation.

8.3 Acceptance Problems and Encoding

We discussed already that a Turing machine is more powerful than FAs or PDAs. We reformulate these concepts in terms of **acceptance problems** which we introduce starting with FAs.
First, we define the following language:

$$A_{FA} = \{\langle B, w\rangle | \text{B is a FA that accepts w}\} \quad .$$

The operator $\langle . \rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$ for some alphabet Σ is an **encoding**. This means that we transform somehow the finite automaton B with input string w into a string (we will see later how). The language A_{FA} contains therefore the encodings of all the FAs with input strings that they accept.
The problem we would like to solve is if there exists a Turing machine that decides A_{FA}, i.e. if the machine halts on every input and accepts exactly the strings that belong to the language.
In other words, is there a Turing machine which given any input string encoding a FA B and a word w, always halts and answers yes or no?
Before trying to solve such problems, we have to better understand what an encoding is. We have to found a way for transforming a FA (in this specific case) into a string or equivalently a number. There are infinite possible encodings and here we propose a simple one.

1. The number of states |Q| can be encoded as a numbers: $q_1 \to 1$ $q_2 \to 2, ...,$
2. Encode the alphabet over Σ (assumed in some lexicographic order) again with integer numbers,
3. Encode the start state with the corresponding number (taken from point 1.).
4. Encode the accepting states with a list of corresponding numbers (taken from point 1.)
5. Encode the transition function $Q \times \Sigma \to Q$ with three numbers already assigned from $Q and \Sigma$.
6. Design a character or combination of characters for separating strings, e.g. # .
7. Append all the previous strings interleaved by # .

After the encoding of the FA B, we can write $\langle B, w\rangle =$ "Encoding of B"# w.
After this procedure, we produced a string representing a FA with an input: this string can be fed into a Turing machine. A similar encoding can be realized also for push-down automata.

How does encoding work in the case of push-down automata, which are non-deterministic for being able to process context-free languages? Actually non-determinism does not prevent encoding but it only affects how acceptance is defined and how the Turing machine simulates the automaton. Even though the PDA has an unbounded stack during execution, *its definition is finite*. That's what matters for encoding. Even though the transition function has as image a power set, its description (all the possible transition rules) remains finite. The problem arises with acceptance, which exists only on some computational branches, which may be infinite (and some might contain infinite loops). What we can do is convert the push-down automaton into a context-free grammar and then run the CYK algorithm on it: this can be simulated with a Turing machine. Then the machine can accept or reject according to the CYK result (CYK always halts).

8.4 Decidability and Languages

We know already that the Turing machine is computationally more powerful than the other models we investigated. The following results formalize these conclusions in terms of decidability. What we saw in the latter section is summarized by:

Theorem 8.4.1: Finite Automata Decidability

Given: FA = 'Finite State Automaton', NFA = 'Non-deterministic FA'.

A_{FA} and A_{NFA} are decidable languages.
This means that a Turing machine can decide if a FA/DFA accepts given an input string w.
Since regular expressions are equivalent to FAs, we can also define

$$A_{REX} = \{\langle R, w\rangle | \text{R is a regular expression generating w}\}$$

and A_{REX} is a decidable language.

For context-free languages we have similar results:

Theorem 8.4.2: Context-free Grammar decidability

Given the language

$$A_{CFG} = \{\langle G, w\rangle | GisaCFGgeneratingw\}$$

A_{CFG} is decidable. There is actually an even stronger result:

Every context-free language is decidable.

The previous results formalize what a Turing machine can do, but in the next section we will see an example of problem which cannot be solved and again the encoding idea will be fundamantal.

8.5 The Halting Problem

The **halting problem** is the question of whether there exists a general algorithm that can determine, for any given computer program and input, whether the program will eventually stop or run forever. It was introduced and resolved in 1936 by Alan Turing in his seminal paper written in the context of '**Hilbert's program**, which aimed to place all of mathematics on a firm, **mechanical** foundation. In this work, Turing formalized the intuitive notion of an algorithm introducing his "machine" and then proved that no Turing machine can solve the halting problem for all possible machines and inputs.
His argument was closely related to Gd̈el's incompleteness theorem (proved few years earlier), and showed that there are **fundamental limits to what can be decided by mechanical computation**. The halting problem is central in logic and computer science, demonstrating that undecidability is an intrinsic feature of computation itself, not a limitation of particular formalisms or technologies.
The following theorem formalizes the latter discussion:

Theorem 8.5.1: Turing's Halting Problem Theorem

The language:

$$A_{TM} = \{\langle T, w\rangle | \text{T is a Turing Machine and T accepts w}\}$$

is undecidable.

Proof. The proof proceeds by contradiction, assuming that A_{TM} is decidable. We assume that there is a decider H for A_{TM} which halts (accepting or rejecting) if it gets as input $\langle T, w\rangle$:

$$H = \begin{cases} \text{accepts} & \text{if T accepts w} \\ \text{rejects} & \text{if T rejects w} \end{cases}$$

Now we define another Turing machine D which uses the machine H as a subroutine:

D = "On input $\langle T\rangle$ (where T is a Turing machine):
1. Run H on input $\langle T, \langle T\rangle\rangle$
2. Output the opposite of what H outputs (accept $\leftrightarrow$ reject)

Note $\langle T, \langle T\rangle\rangle$: you run a machine on itself as an input.
The machine D does this:

$$D(T) = \begin{cases} \text{accepts} & \text{if T rejects } \langle T\rangle \\ \text{rejects} & \text{if T accepts } \langle T\rangle \end{cases}$$

Now let's see what happens if D gets itself as input:

$$D(\langle D\rangle) = \begin{cases} \text{accepts} & \text{if D rejects } \langle D\rangle \\ \text{rejects} & \text{if D accepts } \langle D\rangle \end{cases}$$

This leads to a contradiction: if D accepts, it is forced to reject and vice-versa. □

We propose another proof which is more "condensed" and more "computer-like". Let's say that a Turing machine T is a computer which can take a program p as input: $T(\bar{p})$. The bar over p indicates that we consider the encoding of p. Let's introduce also an operator x which works like a kind of negation: if a machine accepts, the operator makes it reject and vice-versa. We define T as a machine which works like this:

$$T(\bar{p}) = xp(\bar{p})$$

The "program" p is also a Turing machine so we can encode it and give it as an input to itself. Note that this happens also in real situations: a `C` compiler can be written in `C`! And now finally we consider:

$$T(\bar{T}) = xT(\bar{T})$$

But this is a contradiction because if T accepts the latter formula says that it must also reject. Therefore the operation x is nmot possible and the halting of T cannot be decided.

Thinking about this more carefully, we could substitute x with some other operator f and the conclusion would be the same:

$$T(\bar{T}) = f\left(T(\bar{T})\right)$$

This is a case of Curry's Fixed Point Theorem, which we do not discuss further here.

8.6 A 2300-years old Strategy

The **liar paradox** is usually traced back to ancient Greece, around the 4th century BCE. It is traditionally attributed to **Eubulides of Miletus**, a philosopher of the Megarian school, who formulated paradoxes to probe the limits of logic and meaning. The classic version is the sentence

"This statement is false."

If it is true, then it must be false; if it is false, then it must be true!
A variant is the so-called Epimenides paradox, named after **Epimenides of Crete** (6th century BCE), who reportedly said

"All Cretans are liars.".

This sentence seems innocuous, but said by a Cretan, it starts to generate problems. Throughout antiquity up to the Middle Ages, the paradox was discussed by philosophers and theologians concerned with truth, language, and divine omniscience. In the late 19th and early 20th centuries, the paradox came back in the foundations of mathematics and logic.

Russell's paradox and Gödel's incompleteness theorem can both be seen as highly refined descendants of the liar paradox, translated from natural language into formal systems.

A central characteristic of the paradox's sentence is its self-reference. Notice that in Turing's proof there is something similar: a Turing machine that has itself as input. The same "trick" happens in the proof of the Gödel's theorem: a sentence says of itself that it is not provable.

$$R = \{x \mid x \notin x\}$$

Asking whether $R \in R$ leads to a contradiction:

$$R \in R \implies R \notin R, \quad R \notin R \implies R \in R$$

Lier's Paradox	"This statement is false".
Russels's Paradox	Does the set which does not belog to any set belog to itself?
Gödel's Theorem	"This sentence is not provable".
Turing's Theorem	This algorithm does not halt.

Table 8.1: The possibility to realize self-references in different formal systems.

8.7 Rice's Theorem

Rice's theorem says that every **nontrivial semantic property** of programs is undecidable.
These "properties" refer to Turing machines (or programs) that depend only on the function they compute, not on how they are written.
The key is the word *"semantic"*. A semantic property is something like "this program halts on all inputs", "this program computes a total function", "this program outputs 0 on input 1", "this program computes a sorting function over an array". These are properties about behavior or meaning, not about syntax. In contrast, purely *syntactic properties* like "this program has at most 10 states" or "this code contains a certain instruction" are decidable.
Basically, since "this program halts on a specific input" is a semantic property, the theorem is a generalization of Turing's theorem.
The theorem therefore says that the halting problem is not a special, pathologic case, but quite a common example of undecidability.
Conceptually, Rice's theorem is another descendant of the liar's paradox phenomena we discussed earlier. Instead of self-reference through truth or provability, we have self-reference through computation: programs that reason about programs. The theorem tells us that once a system is expressive enough to describe computation in general, it becomes impossible to algorithmically classify programs by their meaning in any nontrivial way.
Another general concluson is that there is no general automatic method to understand what arbitrary programs do. Thus this result, like Gödel's incompleteness theorem and Turing's halting problem, represents a fundamental limit result, showing that undecidability is a structural property of computation itself. Here is the formal statement of the

theorem:

Theorem 8.7.1: Rice's Theorem

Let $\mathcal{P}$ be a property of a partial computable function and

- $\mathcal{P}$ is *semantic*: if two Turing machines compute the same partial function, then either both satisfy $\mathcal{P}$ or both do not.
- $\mathcal{P}$ is *nontrivial*: there exists at least one computable function with property $\mathcal{P}$, and at least one computable function without property $\mathcal{P}$.

Then the language

$$L_{\mathcal{P}} = \{\langle T \rangle | \text{ the partial function computed by T has property } \mathcal{P}\}$$

is undecidable.
In short: **every nontrivial semantic property of computable functions is undecidable.**

Proof. We prove the theorem by contradiction and by reduction from the halting problem.
Since $\mathcal{P}$ is nontrivial, there exist two partial computable functions:

- $f : f \in \mathcal{P}$
- $g : g \notin \mathcal{P}$

Let T_f and T_g the two Turing machines computing these two functions.
We assume (by contradiction) that $L_{\mathcal{P}}$ is decidable. This means that there exists a Turing machine D (a "decider") such that for each machine T:

- $D(\langle T \rangle) =$ accetps iff the function computed by T has property $\mathcal{P}$
- $D(\langle T \rangle) =$ rejects otherwise.

Given an arbitrary pair $\langle T, w \rangle$ we have (halting problem) the undecidable:

$$A_{HP} = \{\langle T, w \rangle | \text{T halts on w}\}$$

We construct now a new machine N which on input x:

- Simulates T on input w
- If T halts on w, then simulate T_f (and thus compute f(x)),
- If T does not halt on w, then simulate T_g.

The machine N will therefore work as following:

- If T halts on w, N computes f
- If T does not halt on w, N computes a non-defined function, or a function g without the property $\mathcal{P}$.

This functioning is summarized by (reduction to the Halting Problem):

$$\langle T, w \rangle \in A_{HP} \Leftrightarrow \langle N \rangle \in L_{\mathcal{P}} \quad .$$

Now we arrive to the contradiction:
Using the decider machine D for $L_{\mathcal{P}}$, we could decide A_{HP} by

- constructing $\langle N \rangle$ from $\langle T, w \rangle$,
- running D on $\langle N \rangle$.

But the fact that A_{HP} is undecidable leads to a contradiction. □

Again, Rice's theorem shows that if we try to algorithmically decide a property of what programs compute, we will fail, because such a decision would (implicitly) solve the halting problem.

8.8 The Post Correspondence Problem

The Post correspondence problem is a problem proposed by Emil Post in 1946 and consists in the following:
give two lists of words $a_1, a_2, .., a_N$ and $b_1, b_2, ..., b_N$ of same length N, a *solution* to the problem consists in a sequence of indices $(i_k)_{1 \leq k \leq K}$ for some $K \geq 1$ and $1 \leq i_k \leq N \forall k$, such that:

$$a_{i_1} ... a_{i_K} = b_{i_1} ... b_{i_K} \quad .$$

The problem looks clearer if we imagine to have a collection of domino-line pieces, each of which has two words in it, like for example[1]:

$$\left[\frac{bc}{ca}\right] \quad \left[\frac{a}{ab}\right] \quad \left[\frac{ca}{a}\right] \left[\frac{abc}{c}\right]$$

A solution can be found rearranging all the tiles in this form:

$$\left[\frac{a}{ab}\right] \quad \left[\frac{bc}{ca}\right] \quad \left[\frac{a}{ab}\right] \left[\frac{abc}{c}\right]$$

[1] Taken from Wikipedia.

If you append all the strings on the top, you obtain the same as appending all the strings on the bottom.
At first glance, the problem looks like a finite combinatorial puzzle. The catch is that the problem **does not bound the length of the sequence!**. You can use one or more tiles more than once. You are not asked whether there is a solution of length $\leq K$ but you are asked whether *some* finite sequence exists, of arbitrary length (possibly enormous) and not known in advance. **The search space is therefore infinite since only the condition $K \geq 1$ is specified.** This simple problem ends up to be **undecidable!**
The genius of Post's problem is that it strips computation down to: "Can two growing strings be kept in sync?", so it is simpler than Turing machines, logic, or arithmetics. This means also that undecidability does not require meaning, arithmetic, logic, infinity of symbols, or self-reference, as we have seen before.
The core of the problem is not in the search space being infinite but in the fact that there is not computable invariant that lets you prune the search. In many infinite searches (e.g. chess, arithmetic Diophantine equations, ...), you can often prove impossibility of some path by monotonicity, invariants, or compactness arguments. Post's problem instead is engineered so that local progress tells you nothing about global success of the search because partial matches of the domino tiles give no information about future compatibility. Another way to see the problem is that we have **local rules and a global constraint (global matching).** So, undecidability arises only from a finite alphabet, a finite input, and an elementary statement. How little structure do you need before undecidability appears? Actually almost none: no states, no symbols with interpretations, no arithmetic: just equality of strings! After these comments about the motivation for inventing such a problem, we have the following theorem

Theorem 8.8.1: PCP Undecidability

The Post Correspondence Problem (PCP) is undecidable.

Proof. The logic of the proof is the following:

1. Given a Turing machine M with input w, we can turn it into an instance $P(M, w)$ of PCP. This is actually the central part of the proof: linking a Turing machine to PCP.

2. $P(M, w)$ has a solution iff M halts on w.

3. If PCP were decidable, we can decide if M halts (M is a PCP decider).

4. This leads to contradiction since the halting problem is undecidable.

The core of the proof is constructing a Turing machine which represent an instance of PCP. The idea is to encode the strings of a domino tile (let's keep using this analogy) with the configuration (state, tape content, head position) of the Turing machine. The full instance (sequence of tiles) will be encoded as a "computational history" of the machine, i.e. the sequence of configurations traversed during the computation. The rest of the proof is mostly technical (but clever). Here we follow the proof of M. Sipser, which is conveniently divided in parts.
Sipser constructs, for any Turing machine M and input w, a PCP instance whose solutions correspond exactly to valid accepting computational histories of M on w. Equivalently, he describes constraints (the "parts" described below) such that any candidate PCP solution must encode a valid Turing-machine computation.

If PCP were decidable, we could decide whether M accepts w, deciding the halting problem. Therefore PCP is undecidable. In other words: PCP is undecidable because it can recognize accepting computations even without simulating them.
Let' see how the construction works:

PART 0: First, for simplicity we modify the PCP problem into a modified one: MPCP. In MPCP the match always starts with the first domino tile. This constraint simplifies the proof but we will recover PCP at the end.

The problem is how to construct an instance P' of MPCP such that it is equivalent to a computational history of M while accepting w.
We define a Turing machine in the usual way $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}\}$
For simplifying the construction we require also: 1. the head does not move left of the starting position, 2. if $w = \epsilon$ we use the string $\sqcup$ instead of w.
Each of the following parts details the functioning of M with input string w.

PART 1: As first domino tile we put

$$\left[\frac{\#}{\#q_0 w_1 w_2 .. w_n \#}\right]$$

where $w = w_1 w_2 ... w_n$, and # a separator symbol. We observe that the first Turing machine configuration ($C_1 = q_0 w$) is written here as a string (this is the central idea)! We have not to somehow extend the top part of the tile (#) for matching the bottom one. The next two parts take care of the transition function.

PART 2: $\forall a, b \in \Gamma$ and $\forall q, r \in Q$ $(q \neq q_{rej})$:

$$\text{if } \delta(q,a) = (r,b,R), \text{ put } \left[\frac{qa}{br}\right] \text{ in P'}$$

PART 3: $\forall a, b, c \in \Gamma$ and $\forall q, r \in Q$ $(q \neq q_{rej})$:

$$\text{if } \delta(q,a) = (r,b,L), \text{ put } \left[\frac{cqa}{rcb}\right] \text{ in P'}$$

PART 4: $\forall a \in \Gamma$:

$$\text{put } \left[\frac{a}{a}\right] \text{ in P'}$$

This rule copies in the top string the same symbols on the bottom, realizing the matching (see example later).

PART 5: This step is to terminate the PCP solution properly. It introduces special ending tiles that can only be used when an accepting configuration has appeared. Once these tiles are used, no further tiles can be appended without breaking the equality.

$$\text{put } \left[\frac{\#}{\#}\right] \text{ and } \left[\frac{\#}{\sqcup\#}\right] \text{ in P'}$$

PART 6: $\forall a \in \Gamma$:

$$\text{put } \left[\frac{aq_{acc}}{q_{acc}}\right] \text{ and } \left[\frac{q_{acc}a}{q_{acc}}\right] \text{ in P'}$$

This part helps the top string "catch-up" with the bottom one which otherwise will be always longer.

PART 7: To "close" the sequence we add the final domino tile:

$$\left[\frac{q_{acc}\#\#}{\#}\right]$$

We have now a procedure for constructing an instance P' of MPCP which exactly simulates the computation of a Turing machine on input w.
Now we should transform P' in the instance P of PCP and this is done introducing the new notation:

$$\begin{aligned} *u &= *u_1 * u_2 * \ldots * u_n \\ u* &= u_1 * u_2 * \ldots * u_n* \\ *u* &= *u_1 * u_2 * \ldots * u_n* \end{aligned}$$

If the MPCP instance is

$$P' = \left\{ \left[\frac{a_1}{b_1} \right], \dots, \left[\frac{a_k}{b_k} \right] \right\}$$

we le the PCP one be

$$P = \left\{ \left[\frac{*a_1}{*b_1*} \right], \left[\frac{*a_1}{b_1*} \right], \left[\frac{*a_2}{b_2*} \right], \left[\frac{*a_3}{*b_3*} \right], \dots, \left[\frac{*a_k}{b_k*} \right], \left[\frac{*\Diamond}{\Diamond} \right] \right\}$$

If P is really an instance of PCP, we are forced to use the first tile in the list above as first tile of the solution, since it is the only one starting with a '*' above and below. The rest of '*' symbols do not disturb any combination.
The last tile in the list containing the new symbol $\Diamond$ just closes the sequence and adds the final extra '*'. Having converted P' in P, the proof is complete. □

8.9 The Busy Beaver Function

This function waas introduced by Tibor Radó in 1962 and it is a concrete example of uncomputable function. In particular the Busy-Beaver [2] function $BB(n)$ is a function $f : \mathbb{N} \to \mathbb{N}$ which grows *faster* than any other function.
This is the definition of such incredible function:

[2]The name for this function was given by Rado himself, picturing a beaver busy jumping back and forth on the Turing machine's tape.

n	BB(n)	Year of Calculation / Authors
1	1	Obvious
2	6	1963, Lin
3	21	1963, Lin
4	107	1983, Brady
5	47,176,870	2024, The Busy Beaver Challenge
6	$> 10 \uparrow 15$	2010, Kropitz
7	$> 10^{2\times 10^{10^{10^{18,705,352}}}}$	2014 "Wythagoras"
8	?	?

Table 8.2: Known values and lower bounds of the Busy-Beaver Function.

Busy Beaver Function

Let M be a Turing machine with alphabet $\{0, 1\}$ and a two-way tape (so that the machine's head can move left or right).
Given a machine T, let $s(T)$ the number of steps the machine performs before halting, including the final `HALT` step. The starting configuration is an all-zero tape. If M never halts, $s(M) = \infty$.

Let T(n) the set of all Turing machines with n states. $|T(n)| = (4n + 1)^{2n}$. In is interesting to note that the binary description of a machine grows like $n \log_2 n + O(n)$ bits.

The **Busy-Beaver Function BB(n)** is defined as:

$$BB(n) = \max_{M \in T(n): s(M) < \infty} s(M)$$

This means that among all the $|T(n)|$ Turing machines, disregarding the ones that do not halt, the n^{th} Busy-Beaver number $BB(n)$ is the largest number of steps done.
The particular machine which does the maximum number of steps is called the **n-state Busy Beaver**.

At first sight this function is perfectly defined, but as we said before, it grows faster than any other function. This means that given a function over the integers $f(n)$, after a certain n we always have $BB(n) > f(n)$.
The Busy-Beaver function grows so incredibly fast that we know only a few of its values (see Tab. 8.2). The fastest-of-all property is formalized in the following theorem:

Theorem 8.9.1: Busy-Beaver Fastest Growth

Assume $f : \mathbb{N} \to \mathbb{N}$ a computable function.
Then, $\exists n_f \in \mathbb{N}$ such that

$$BB(n) > f(n) \quad \forall n > n_f$$

Proof. We assume a Turing machine M_f with c states which calculates f on any input. Now we define a second machine N which can simulate BB and run longer than M_f.

- Remember than BB starts on an empty tape, so we have to encode the input to f somewhere. So N, on blank input, must write a number n on his tape.
- Simulate M_f to compute f
- After computing $f(n)$, perform exactly f(n)+1 (or $f^2(n)$, or ..) dummy steps (a sort of "empty loop"): this converts the value of f into a lower bound on time.
- `HALT`.

The total number of states of N is $s(n) = c + \log(n)$ where c is the number of states needed for computing f and $\log(n)$ encodes the input (in binary).
By definition of BB (BB returns the maximum number of steps):

$$BB(c + \log(n)) > f(n) \quad \forall n \quad .$$

Since given the monotonicity of BB, for some $n > n_f$: $BB(n) \geq BB(c + \log(n))$, we have

$$BB(n) > f(n) \quad n > n_f \quad .$$

An alternative, simpler proof is the following, by reduction to the halting problem:

Assume by contradiction, that there is a function f such that $f(n) > BB(n)$ for all $n \in \mathbb{N}$.
Then, given a Turing machine T, we can first compute $f(n)$ and then simulate T for up to $f(n)$ steps.
If T has not halted after computing f, then it will never halt, because $f(n)$ is larger than $BB(n)$, i.e. the largest possible number of steps an n-state machine can make.
This gives us a way to decide the Halting Problem, which we know it is impossible (contradiction). Therefore, f cannot exist. □

Note that there are infinitely many s such that $BB(n) > f(n)$. The fact that BB grows faster than any computable function rises the strong suspect that BB is not computable, and indeed:

Theorem 8.9.2: BB is uncomputable

The Busy Beaver function $BB(n)$ is not computable.

Proof. - Assume by contradiction that BB is computable.
- Then BB is a total computable function (computable on each possible input).
- But as seen before, BB grows faster than any computable function.
- Then BB should grow “faster than itself”, leading to a logical contradiciton.

Hence BB is not computable.
In other words, BB is uncomputable not merely because it grows fast, but also because if it would be computable, it would outrun itself! □

8.10 Turing Degrees

We conclude this chapter with some comments about the general class of problems solvable by Turing machines trying to understand the relationships between different problem and looking for a classification.
We start giving the definition of an **oracle**. An oracle (apparently introduced by Turing in his PhD thesis) is a theoretical “black-box” able to deliver the correct solution to a problem (even a very difficult one or even an impossible one like the halting problem). Having an oracle, we can build the following definition:

Turing Reducibility

Given two problems A and B,
A is **Turing-Reducible** to B if A is solvable by a Turing machine *given* an oracle for B.

The definition basically says that if we have a “magic device” able to solve B, then we can solve A. This means that A is *not harder* than B. If this property is true also in the other direction we have

Turing Equivalence

If two problems A and B if each is Turing-Equivalent to the other.

The previois definitions induce a general classification of the problems in equivalence classes. Each class has a **Turing Degree**: A Turing degree is the set of all problems which are Turing-equivalent to a given problem.
Two degrees are simple to identify (actually, we did it implicitly before):

- The set A of computable problems.
- The set B of problems equivalent to the halting problem.

Since the halting problem is not computable, $A \cap B = \emptyset$.
These two classes were easy to identify, but two more questions arise naturally:

1. There is a class "above" B? (Something even more difficult).
2. What about the existence of classes *between* A and B?

The answer to question 1. is not too difficult: we can consider an improved version of the halting problem:

Given a Turing Machine T with a halting-problem oracle, decide if T halts.

This is just a "shifted-up" version of the original problem and it indeed more difficult (or has higher Turing-degree, or it is not reducible to the "simple" halting problem).
Quesion 2. is more complex, and it was asked for the first time by Emil Post in 1944. The problem was solved in 1954 by Post himself and Stephen Kleene in a positive way. Such "intermediate" problems exist, but are extremely contrived: nobody found yet a "natural" problem in this class.
Given two such problems A and B, both are solvable given an oracle for the halting problem but no one can be reduced to the other.

9 (Quantum) Computation in the Physical World

9.1 Introduction

Through the Extended Church-Turing Thesis, computation enters the real world of physical systems. Concretely, every computation is executed by a physical system which obeys some laws (at least the current known laws of physics!). In this sense, physical laws are connected to computation: knowing the physical laws can shed light on the limits and capabilities of computation, but an intriguing idea is also that studying computation might say something about the physical world.

10 Conclusion

Grammar/Language	**Recognizer / Machine**	**Time**	**Space**
Regular	Finite automaton	$O(n)$	$O(1)$
Context-free	Pushdown automaton	$O(n^3)$ (general)	$O(n)$
Context-sensitive	Linear-bounded automaton	PSPACE-complete	$O(n)$
Recursively enumerable	Turing machine	Unbounded	Unbounded

Table 10.1: Chomsky hierarchy: grammar types, languages, recognizers, and computational complexity.