ON THE PHENOMENON OF LIFE

Yehan Hathurusinghe

16-Jan-2026
Hash:817c325febb7b121357fe53d92594abbe053f4616987f3170687bd2c8ed5567d

Abstract

*This monograph seeks to justify the interpretation of the universe as a form of computational system and of life as one of its programs, specifically a self-replicating and evolving process comparable to Cohen's theory of computer viruses and von Neumann's universal constructor. Upon further analysis, it is argued that this computational characterization is not a direct property of physical reality, but rather a consequence of the generalized computational framework required for human explanation. All explanations of reality are fundamentally symbolic, taking the form of strings, and consequently acquire the structural features of executable descriptions analogous to computer programming codes. This necessity reflects the nature of explanation, not the requirement that the universe itself be a computer. Nevertheless, nothing precludes modeling the universe computationally, nor interpreting physical events as programs within such a framework; indeed, this may be the only coherent approach available. Building on the foundational work of Turing, von Neumann, Chaitin, and Cohen, the monograph introduces **Generalized Computational Framework**, within which the universe may be consistently viewed as a computational system and its processes as programs. Despite its conceptual simplicity, the framework appears to address a broad range of central philosophical problems.*

Preface

I did not devote the entirety of past five years to the origin-of-life problem. Records indicate that I first began collecting research on the subject in 2021. My interest deepened when I encountered the bacterial flagellar motor, a phenomenon that profoundly impressed upon me the sophistication of living systems. I later chose this topic for a biophysics presentation in a course taught by Professor Anton Zilman, who too shared my fascination.

Up to 2023, I studied theoretical work by Günter Wächtershäuser, William Russell, and Michael Martin on the iron–sulfur world hypothesis, as well as the hypercycle theory developed by Manfred Eigen and Peter Schuster. In parallel, I followed experimental research on the RNA world by Jack Szostak and by David and Bruce Damer, along with work by younger researchers producing synthetic ribozymes. I was also deeply fascinated by the Miller–Urey experiment, first conducted in the 1950s, and by its many modern variants, which continue to yield promising results. Throughout this period, I found inspiration in episodes of Carl Sagan's Cosmos, which accompanied many of my days. At the same time, I pursued a separate project on the mechanisms of economics, motivated in part by concerns about financial stability, and undertook the necessary self-study in cell biology, chemistry, and biochemistry to properly understand the research literature.

It was during the summer of 2024 that I began to seriously associate computational theory with life. Although I had long been aware of connections such as Conway's Game of Life, it was a careful study of von Neumann's universal constructor that proved decisive. I was struck by the correspondence between von Neumann's instruction tape, tape reader, constructor arm, and universal copier, and the biological machinery of DNA, RNA polymerase, ribosomes, and replication processes. At this point, I became deeply engaged with computational theory.

Viewed through this lens, the universe can appear as a kind of computer, within which life functions as a self-replicating and evolving program, analogous to computer viruses of Fred Cohen, and the Universal Constructors of von Neumann. However, I soon recognized that such a view, if left vague, could not constitute a scientific theory. This led me to the realization that it is not life itself that resembles a virus, but rather the description of life. I came to distinguish between the universe as it exists and the universe as it is described by human consciousness. What connects the two is that descriptions of the universe take the form of strings, which are mathematical objects upon which formal operations can be performed.

From this observation, I developed what I call **Generalized Computational Theory**. In this framework, phenomena and their descriptions are treated as distinct, with descriptions serving as inputs to a hypothetical generalized computer capable of recreating phenomena. This idea extends the conceptual lineage of the Turing machine and draws inspiration from Alan Turing's work on oracle machines, with oracles generalized to account for matter and consciousness while remaining describable as strings. The theoretical work presented here reflects only the initial stage of this effort; most of the preceding years were spent identifying the appropriate mathematical foundations, and future volumes will develop the framework in much greater detail.

Acknowledgments

1 Life: An Extraordinary Phenomenon out of Ordinary Matter

Life is a remarkable phenomenon unique to Earth, distinguished by its extraordinary complexity and unparalleled capabilities that have fascinated and challenged scientists and philosophers for centuries. Unlike inanimate matter such as minerals, rocks, planets, and stars across the observable universe, life exhibits unique traits, including the ability to transport matter, grow and develop, reproduce, maintain homeostasis, evolve to adapt and thrive in diverse environments, and give rise to intelligent agents capable of rational thought, emotional responses, and purposeful interactions with their surroundings. These attributes set life apart as an extraordinary exception in the cosmos.

The basic unit of life is the cell, a complex structure equipped with internal machinery capable of self-replication and maintaining its structural integrity while enveloped by a dynamic membrane that interacts with its external environment to sense, transform, or adjust to changes. Life exists either as unicellular organisms, functioning as single agents, or as multicellular organisms, where cells are organized into specialized components. Through metabolism, life converts simple compounds such as carbon dioxide, nitrogen, phosphates, sulfates, and mineral compounds, into the matter that constitutes its body. While not all life forms develop organs enabling movement across their environments—examples include plants and some unicellular organisms—all cells contain internal structures that employ motion for intricate tasks such as particle transport, reproduction, and metabolism. Notably, motion is not a prerequisite for the function of all complex objects; for example, computers can operate without complex physical movement, relying solely on electron flow through electric currents. However, motion—whether in the form of classical mechanics or quantum transport—remains a fundamental aspect of life, playing a critical role in its essential processes.

All cells are composed of lipid membranes, proteins, RNA, DNA, coenzymes, miscellaneous sugars, and metallic ions. Lipid membranes form the cell envelope, internal compartments, cellular organelles, and vesicles used for transport. Proteins consist of peptides, which are polymers of amino acids, while RNA and DNA are polymers of ribonucleotides and deoxyribonucleotides, respectively. Proteins and RNA form cellular machines, including molecular motors, polymerases, and enzymes, while DNA serves as the repository of genetic information. Coenzymes, such as ATP and chlorophylls, function as tools for proteins and RNA, aiding in various biochemical processes. Miscellaneous sugars, such as polysaccharides, provide structural support to membranes, while metallic ions, in addition to acting as coenzymes, play a critical role in energy storage through the formation of electrochemical gradients essential for cellular function.

The universal cellular machinery shared among all cells includes **transcription**, **translation**, and **replication**. DNA, a double-helix structure that stores genetic information, is organized into genes and regulatory regions. Genes encode the components of proteins and RNA machines. When a protein is needed, RNA polymerase transcribes the corresponding DNA segment into RNA, forming messenger RNA (mRNA). This mRNA is then translated by the ribosome, a large RNA-protein complex, into a peptide chain. One or more peptide chains fold to form a functional protein. RNA can also function independently as a ribozyme, while some essential cellular machines, like the ribosome, are composed of both RNA and proteins. During cell reproduction, DNA is replicated into one or more copies by DNA replication machinery, which are then actively transported to specific regions of the cell, ensuring accurate distribution to daughter cells during division.

There are two types of cells: **prokaryotes** and **eukaryotes**, with prokaryotes being the more ancient and primitive of the two. Prokaryotes are further divided into **bacteria** and **archaea**. It is believed that during a process called **symbiogenesis**, an ancient archaea-like cell engulfed an aerobic bacteria-like cell, giving rise to the eukaryotic cell type, with the engulfed bacteria evolving into mitochondria, which serve as the cell's power factory. Later in evolutionary history, eukaryotic cells also engulfed photosynthetic bacteria-like cells, leading to the formation of plant cells. All extant life is believed to share a common ancestor known as the **Last Universal Common Ancestor** (LUCA). This hypothetical organism is thought to have existed billions of years ago, serving as the foundational root from which all current forms of life evolved. LUCA is not considered the first life form but rather the most recent organism from which all living organisms inherited shared molecular and genetic features, such as the genetic code, DNA and RNA-based information storage, and core metabolic pathways.
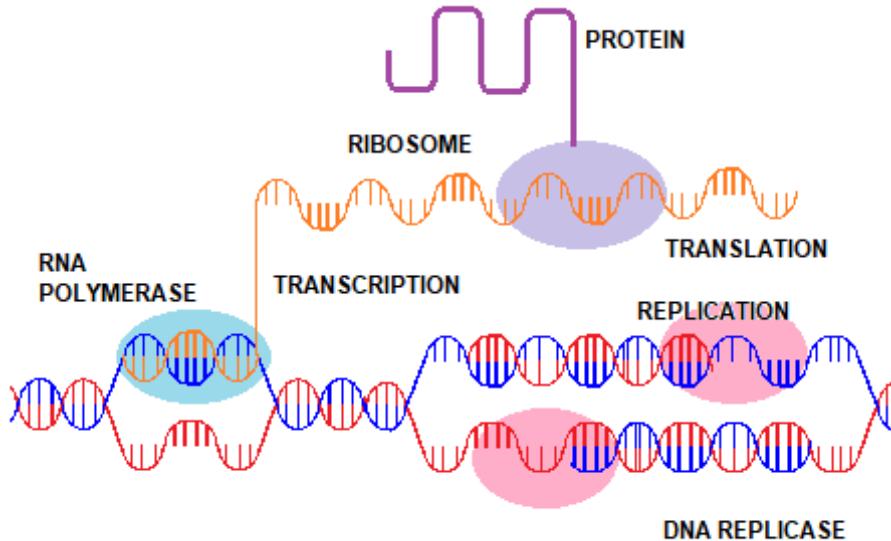
Figure 1: Universal Genetic Machinery of Cells

Metabolism is the process by which nutrients and energy are transported into the cell and transformed into cellular materials, energy carriers, and redox agents. Photosynthesis, for instance, occurs in bacteria like cyanobacteria and in chloroplasts within plant cells. This process uses solar photons to maintain a proton gradient by pumping protons (hydrogen ions) into a compartment of high proton concentration. The gradient powers a motor protein called ATP synthase, which converts ADP to ATP, the primary energy carrier in cells, and reduces NADP+ to NADPH, which participates in redox reactions such as the Calvin cycle. The Calvin cycle assimilates carbon dioxide into glyceraldehyde-3-phosphate (G3P), the building block of glucose. Similarly, nitrogen, sulfur, and phosphates, as well as metals, are assimilated to produce coenzymes, amino acids, and nucleotides. Nucleotides are synthesized through a complex biosynthetic pathway involving amino acids, phosphorylated sugars derived from glucose, and inorganic phosphate.

Different genetic expressions give rise to distinct cell types, and the genome defines a cell's species. In multicellular organisms, all cells typically share the same genome, yet they differentiate into specialized types through the regulation of gene expression. This process is controlled by complex networks of transcription factors, epigenetic modifications, and signaling pathways that activate or suppress specific genes in response to developmental cues and environmental signals.

Cell division in bacteria is an intricate process that involves the formation of a Z-ring and the active transport of the DNA chromosome. The Z-ring, acts as a scaffold to recruit other proteins essential for cytokinesis, ensuring the bacterial cell divides accurately into two daughter cells. Additionally, bacterial DNA chromosomes are not randomly organized; instead, the regions of the DNA are spatially aligned with their functional roles in specific regions of the cell. This organized arrangement, known as nucleoid organization, facilitates efficient transcription, replication, and segregation during cell division, highlighting the remarkable coordination underlying bacterial cellular processes.

In eukaryotes, lipid membranes are utilized to form various structures such as internal organelles and compartments, with their formation and function being highly coordinated by proteins that bind and interact with them. Alongside these membranes and the organization of the DNA genome into multiple chromosomes, the existence of a dynamic cytoskeleton has had a revolutionary impact on eukaryotic cell structure, serving as a foundation upon which all other cellular complexities depend. The cytoskeletal proteins in eukaryotes are believed to have bacterial ancestry, evolving into the intricate structural network observed today.
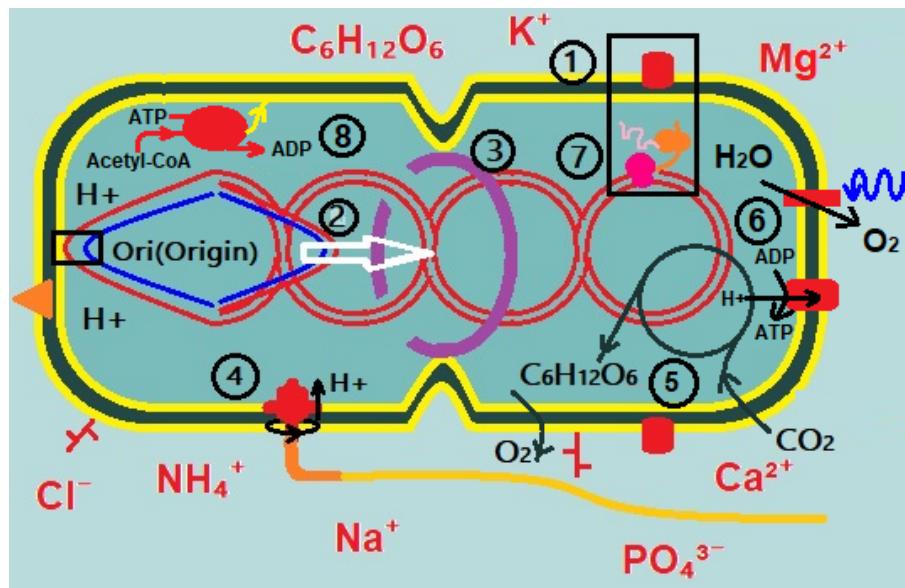
Figure 2: Common Prokaryotic Cell Structure and Function: *(1)Inter-membrane space proton gradient. (2)DNA replication, and chromosome segregation through ATP-powered active transport. (3)Z ring formation. (4)Flagella. (5)Metabolism. (6) Photosynthesis and ATP synthesis*

Transcription in eukaryotes occurs within a specialized nucleus, where the DNA is transcribed into RNA. This RNA is transported out of the nucleus into the endoplasmic reticulum (ER), a membranous organelle where translation occurs via ribosomes attached to its surface. The resulting proteins are packaged into vesicles and transported to the Golgi apparatus, another membranous organelle. In the Golgi, proteins undergo modifications such as glycosylation, phosphorylation, and sorting, ensuring they are functionally mature and correctly tagged for their destinations. The Golgi then generates more specialized vesicles containing the proteins, which are transported to various regions of the cell. This intracellular transport is facilitated by motor proteins such as dynein, kinesin, and myosin, which move along the cytoskeletal filaments: **microtubules**, intermediate filaments, and **actin** filaments. These motor proteins "walk" along the filaments, powered by ATP, carrying vesicles to their destinations with high specificity. As the vesicle approaches its target, the proteins on its surface interact with complementary proteins at the destination site, ensuring precise docking and delivery of the cargo. This dynamic transport system enables eukaryotic cells to maintain their intricate and highly organized structure, supporting complex processes like signaling, secretion, and cellular motility. Such organizational sophistication is particularly evident in protozoa, which exhibit remarkable structural and functional complexity despite being unicellular organisms. In animal cells, the hyper-utilization of myosin and actin has given rise to **muscle cells**, while the extensive adaptation of microtubules has contributed to the development of **nerve cells**.

Genetic recombination and exchange play a crucial role in generating diversity in life. Processes such as crossing over, independent assortment, and horizontal gene transfer introduce genetic variation, enabling populations to adapt and evolve. In prokaryotes, genetic exchange occurs through transformation, transduction, and conjugation. During transformation, a bacterium takes up free DNA from its environment, which may integrate into its genome and lead to new traits. Transduction involves bacteriophages (viruses that infect bacteria) delivering foreign DNA into a bacterial cell. Conjugation enables direct transfer of genetic material—often plasmids—between bacterial cells via a pilus. In eukaryotes, genetic recombination occurs during meiosis in sexual reproduction, producing haploid gametes (sperm and eggs). Crossing over (in prophase I) and independent assortment increase genetic variability by shuffling alleles. Upon fertilization, haploid gametes from distinct parents fuse to form a diploid zygote, restoring the chromosome number and producing offspring with unique genetic combinations. This genetic diversity is essential for evolution and adaptability.
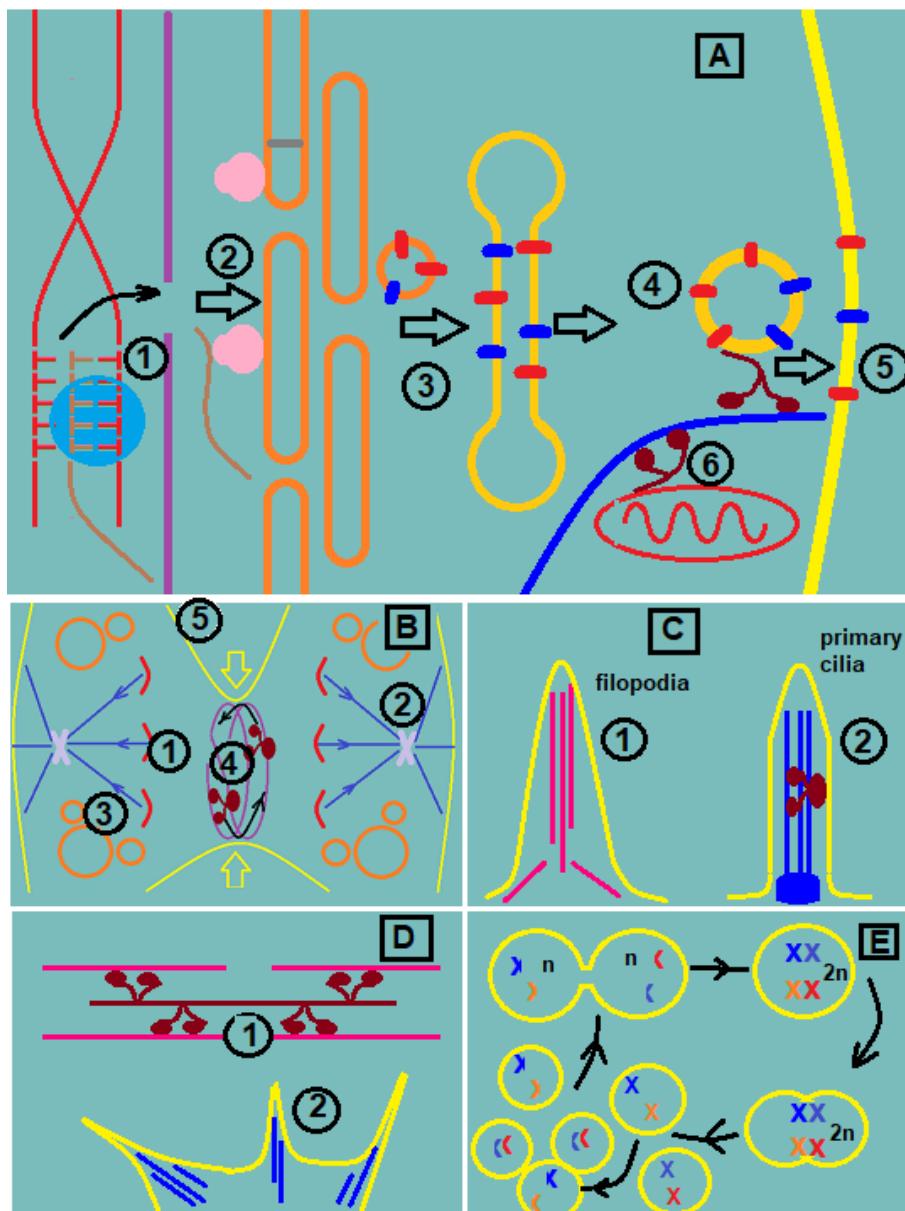
Figure 3: (A) Eukaryotic Protein Synthesis, Transport, and Dynamic Membrane Regulation: *(A1) RNA transcription within the nucleus, (A2) Active transport of mRNA into the cytoplasm and to the Endoplasmic Reticulum, where translation by ribosomes and protein synthesis occur, (A3) Vesicular transport of proteins to the Golgi apparatus and sorting into target vesicles, (A4) Transport of vesicle-protein complexes by motor proteins, (A5) Assimilation of vesicle-protein complexes at the membrane target regions, and (A6) Mitochondrial transport mediated by motor proteins.* (B) Eukaryotic Cytokinesis and Chromosome Segregation:*(B1)Active chromosome tranxport by spindlex. (B2) Centrosome. (B3) Cell organelles disassembly in to vesicles. (B4)Contraction ring mechanism involving motor proteins. (B5) Cleavage furrow.* (C) Filopodia and Cilia *(C1)Actin filaments form the core of filopodia. (C2) Microtubles form the core of of cilia.* (D) Eukaryotic divergence in to Muscle and Neurons. *(D1) Actin and Myosin in muscular mechanism. (D2) Microtubles form the Dendrites in Neurons.* (E) Mitosis (2n → 2n): *One cell divides into two identical cells, maintaining chromosome number.* Meiosis (2n → n): *One cell divides twice to produce four non-identical cells, each with half the chromosome number.*

Life can be envisioned as a machine with two complementary hands: one constructed from proteins and the other from nucleic acids. These hands work together to synthesize both proteins and nucleic acids as new components of these hands, which carry out metabolic, regulatory, and motor functions throughout the cell, orchestrating the fundamental processes of life. While a single hand can perform numerous

tasks independently, their cooperative interplay achieves far greater outcomes. This duality may echo the origins of life, where the RNA "hand" likely played a pivotal role in shaping more proficient peptide "hands", which subsequently facilitated the development of more advanced RNA "hands".

Human civilization has been constructed through the use of human hands—building structures and machines with tools that were themselves crafted by human hands and machines, all from materials drawn from the surrounding environment. This process finds a striking parallel in biology, where proteins and nucleic acids serve as the functional "hands" of life, while coenzymes act as the tools through which these molecular hands perform their work.

Sexual reproduction involves the fusion of sperm and egg, where the sperm typically travels to find an egg, often across distances. This process of sperm reaching the egg, whether across vast distances in animals or within a localized area in plants, is a beautiful and intricate part of reproduction that ensures genetic diversity and the continuation of species. Sexual reproduction might have originated when slightly mutated cells of the same species fused together as they formed rudimentary multicellular organizations, resulting in the mixing of their genetic material.

The origin of animals can be seen in sponges, which provide a simple yet foundational example of early multicellular life. In sponges, Archaeocytes, or amoebocytes, are totipotent cells that can transform into various other cell types. They digest food, transport nutrients, and produce eggs for reproduction. Archaeocytes also give rise to specialized cells that maintain the sponge's structure, such as Sclerocytes, which produce calcium spicules. Choanocytes, flagellated cells that line the spongocoel, play a crucial role in both feeding and reproduction. These cells trap food particles with their sieve-like collars and generate a unidirectional water flow through the sponge's body, enabling nutrient capture and waste removal. Additionally, choanocytes can differentiate into sperm cells for sexual reproduction. Together, these versatile cell types illustrate how even the simplest animals developed specialized functions that laid the groundwork for the evolution of more complex multicellular organisms.
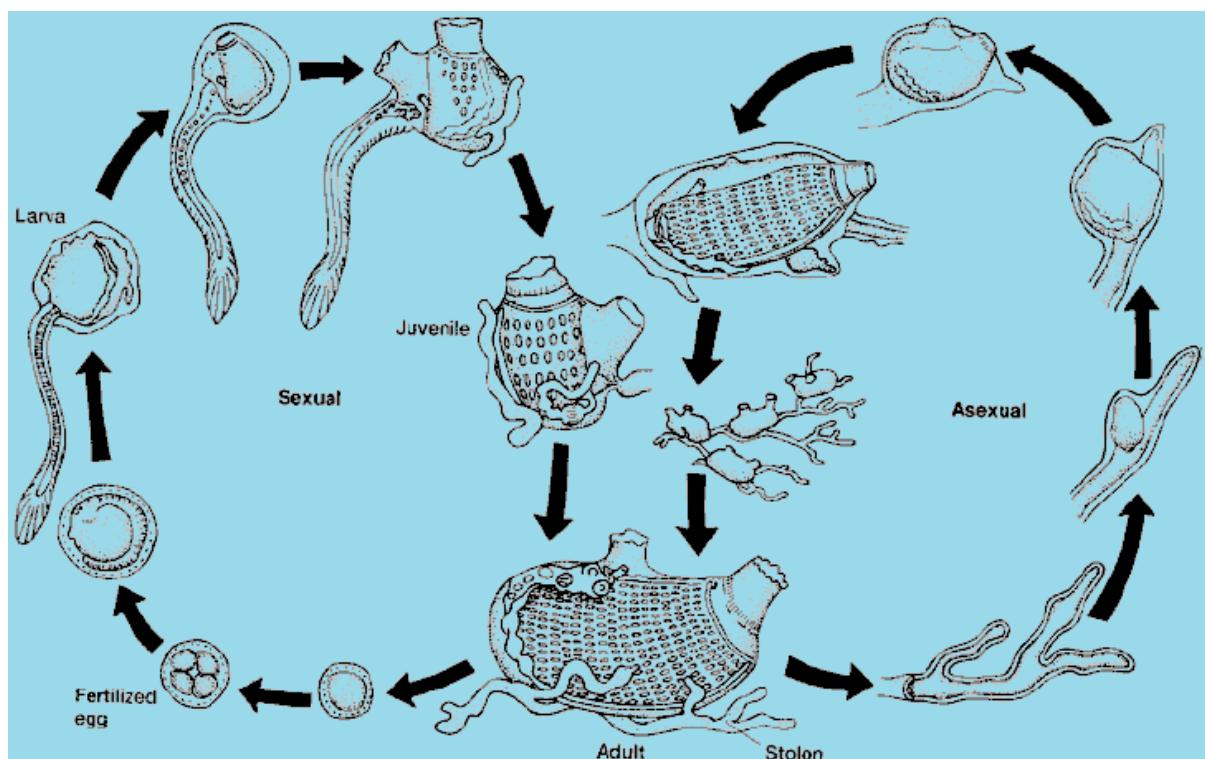
Figure 4: Tunicate or Sea Squirt life cycle: *Sea squirts, or tunicates, belong to the subphylum Tunicata and exhibit a fascinating lifecycle. Their free-swimming larval stage resembles vertebrates, possessing a notochord, dorsal nerve cord, and tail. These features highlight their evolutionary link to modern chordates.* Source: Swalla, Billie. (2006). Building divergent body plans with similar genetic pathways. Heredity. 97. 235-43.

Sea squirts, or ascidians, provide another fascinating example of early animal evolution. Their larvae, which are free-swimming and possess features similar to thoe found in the animal kingdom. The larval form of sea squirts has a notochord, a dorsal nerve cord, and pharyngeal slits, which are all hallmark features of chordates—the group that includes vertebrates. In addition, the larvae contain nerve cells and muscle cells, which enable the larva to swim using a tail powered by muscle contractions. These structures and the primitive nervous system of the larva resemble that of early vertebrates. As the larva matures, it undergoes metamorphosis, settling on a surface and transforming into an adult form that is sessile and has lost many of these vertebrate-like structures.

Life behaves distinctly from other forms of matter, but upon closer examination, it is made up of the same elements found in air and rocks. Through the work of many scientists over the years, it has become clear that both complex multicellular organisms and unicellular organisms have evolved through genetic mechanisms, particularly through random mutations and natural selection. These processes have driven the diversity of life, allowing organisms to adapt to their environments and pass on advantageous traits to future generations.

In the 20th century, scientists began to study the origin of life as a geochemical phenomenon. The Miller-Urey experiment, conducted in 1952, was a groundbreaking experiment that simulated the conditions of early Earth to test the hypothesis that life could have formed from simple chemical reactions. The power of the Urey-Miller experiment lay in its simplicity. By using basic ingredients such as methane, ammonia, hydrogen, and water in specific ratios, and subjecting them to an electric arc to simulate lightning, the experiment successfully produced amino acids and other prebiotic organic molecules. This simplicity is its greatest strength; despite debates over whether Earth's early atmosphere precisely matched the experimental conditions, the fundamental simplicity of the setup remains indisputable. While it's true that Earth's atmosphere may not have been uniformly reducing, with some regions potentially harboring the necessary reducing gases, the core concept and results of the experiment underscore the potential for simple chemical processes to generate the building blocks of life. Many Miller-Urey-like experiments have been conducted since then, and it was found that the boron-silicate composition in the apparatus used played a substantial role as a catalyst. In 2023, another milestone was reached with the publication of a report detailing the prebiotic synthesis of ATP and other crucial building blocks of life. This experiment, reminiscent of the original Urey-Miller experiment, utilized synthetically produced seawater, calcium phosphate , and magnesium sulfate minerals, yielding nucleotide, sugars, peptides, amino acids, fatty acids, and alpha acids.
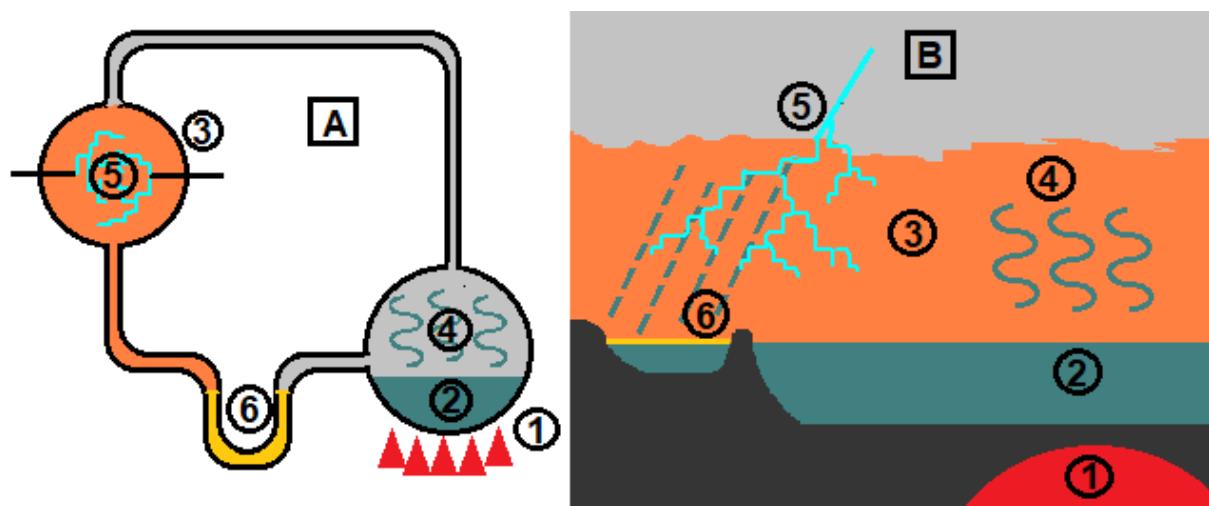
Figure 5: Urey-Miller Experiment: (A) Experimental Setup: *(A1) Heat. (A2) Water. (A3) Reducing gases (ammonia, hydrogen, methane, and water vapor). (A4) Evaporation. (A5) Electric Spark. (A6) U-tube for collecting organic products.* (B) Plausible Prebiotic Hydrothermal Environment on Earth: *(B1) Geothermal Heat. (B2) Vast Ocean. (B3) Reducing Environment. (B4) Evaporation. (B5) Lightning. (B6) Presence of mineral catalysts and the formation and concentration of organic compounds in mineral pores.*

The intricate structure of RNA and DNA, with their well-organized double-strand configurations, might initially appear unusual, yet over time, scientists have discovered molecules with similar behaviors. Notably, synthetic nucleic acids such as xeno nucleic acids(XNA) with modified bases and ribose structures have been found to exhibit parallel structural and functional properties. Additionally, various nucleobase analogs show diverse base-pairing characteristics, broadening understanding of molecular biology and its potential alternatives. Nucleic acids, rather than being inherently special, likely emerged from a variety of prebiotic polymers that were selected for their exceptional advantages. Their unique properties provided evolutionary benefits, leading to their central role in the molecular machinery of life.

In the vast expanse of the solar system, moons like Titan, Saturn's icy moon, offer glimpses into the potential for life beyond Earth. Titan, with its methane clouds, frozen surface, and organic compounds, showcases complex chemical processes vital for prebiotic chemistry, yet lacks essential elements like phosphorus that are crucial for life. While Earth benefits from a unique combination of conditions, including a protective magnetic field, geothermal heat, organic compounds, and vital elements like phosphorus, other worlds like Venus and Mars show varying distributions of essential elements without offering the full conditions necessary for **abiogenesis**. Mars, for instance, once possessed geothermal activity but lost much of its internal heat as its core cooled, eliminating a major energy source for the chemical processes associated with abiogenesis. Earth stands out in the solar system for having all the vital ingredients and conditions required for the emergence of life, with its blend of metals, organic compounds, and minerals. The overwhelming evidence supporting chemical origin of life(abiogenesis) suggests that life, though exhibiting extraordinary behavior, likely originated from the same fundamental elements as ordinary matter, with organo-metallic compounds possibly containing the necessary potential for life's emergence.

For reference: [A1],[A2],[A3]

2 Turing Machines and Computational Theory

Computational theory, a captivating field that has intrigued researchers since the early 20th century, was originally developed to study computers, their applications in areas like cryptography, and their inherent limitations. However, much like calculus and geometry were pivotal in Newton's time for studying celestial motion, computational theory is indispensable for exploring the phenomenon of life.

Alan Turing, widely regarded as the father of modern computer science, made groundbreaking contributions to the understanding of computation through his concept of the **Turing Machine**. Introduced in 1936, the Turing Machine is a theoretical device that models the fundamental principles of computation. It consists of an infinite tape, a read-write head, and a set of rules governing its operations.

Definition 2.1. *A **Turing Machine** M is a 6-tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F),$$

where Q, Σ, Γ are all finite sets and:

1. *Q is the set of states,*

2. *Σ is the input alphabet not containing the blank symbol $\square$,*

3. *Γ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$,*

4. *$\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,*

5. *$q_0 \in Q$ is the start state, and*

6. *$F \subseteq Q$ is the set of halting states.*
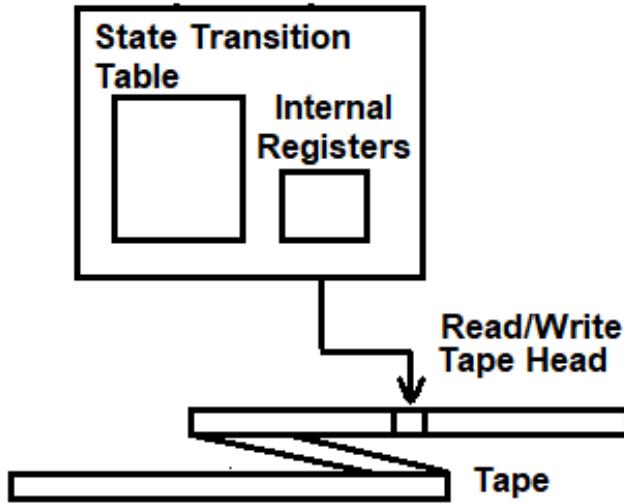
Figure 6: Schematic representation of a Turing Machine: *Shows the most important components, although Turing Machines are a mathematical abstract entity that does not require internal mechanisms*

Definition 2.2 (Language of a Turing Machine). *Let M be a Turing machine. The language recognized by M, denoted $L(M)$, is the set of all binary strings that M accepts:*

$$L(M) := \{x \in \{0,1\}^* \mid M \text{ halts and accepts on input } x\}.$$

Here, M accepts a string x if, when started on input x, the computation eventually halts in an accepting state.

Proposition 2.1 (Enumerating the Language of a Turing Machine). *Let M be a Turing machine with language $L(M) \subseteq \{0,1\}^*$. Then $L(M)$ can be enumerated by the following computable procedure:*

1. *Enumerate all binary strings in $\{0,1\}^*$ as $s_0, s_1, s_2, \ldots$ (for example, lexicographically by length).*

2. *Initialize $n = 0$.*

3. *For $k = 1, 2, 3, \ldots$ (step counter):*

    (a) *For each $i = 0, 1, \ldots, k$, simulate M on input s_i for one additional step beyond the previous simulations.*

    (b) *If M halts on s_i during this simulation, output s_i as the next element of the enumeration of $L(M)$.*

4. *Repeat indefinitely.*

This procedure ensures that every string in $L(M)$ will eventually appear in the enumeration.

An **algorithm** is a step-by-step procedure for manipulating a sequence of symbols according to a well-defined set of rules, in order to produce a new sequence of symbols. According to the **Church-Turing Thesis**, any such algorithm can be implemented as a Turing Machine. This thesis asserts that anything **computable** by an algorithm can also be computed by a Turing Machine, essentially providing a formal framework for the notion of **computability**. Turing Machines therefore separate computations from human mathematicians and treat them as mathematical objects by themselves from which properties can be studied. By formalizing computation, Turing Machines abstract away the need for human involvement in each step, transforming computation into a purely mathematical process. Instead of considering computation as a task requiring human intervention and creativity, Turing Machines treat it as an object of study in its own right. This abstraction allows for the examination of computational properties, such as decidability, complexity, and computability, independent of human intuition or specific algorithmic implementations.

The term "program" has been increasingly misused in the commercial computing world, where it is often conflated with the code entered into computers as input. Commercial computer programmers frequently fail to distinguish between the program itself and the input code. However, within theoretical literature, "program" holds a more profound and precise definition. A program is understood as the entire behavior of a computational machine as it operates on specific input instructions. For example, a Turing program is not merely the input string but encompasses the entire machine and its behavior when the string is provided as input.

Definition 2.3. *A **Turing program** is a pair (M, w), where M is a Turing Machine and w is an input string. The Turing program is understood to be the behavior of M as it processes w according to its transition function, including all intermediate tape configurations and state transitions. A **Turing program set** is defined as a pair (M, W), where W is a set of input strings to M.*

A **Multi-Tape Turing Machine** is a Turing Machine that uses multiple tapes, each with its own tape head. These heads can operate independently on the respective tapes, allowing for parallel processing of information. A **Multi-Head Turing Machine** refers to a Turing Machine that uses multiple tape heads on a single tape. Each head can move independently and read/write to the tape, enabling the machine to process different parts of the tape simultaneously. A **Networked Turing Machine** is a collection of Turing Machines that are connected together in some way. These machines can communicate or share information, with each machine performing parts of the overall computation.

According to the Church-Turing thesis, despite structural differences, **Multi-Tape**, **Multi-Head**, and **Networked Turing Machines** are all **equivalent** in terms of computational power to a standard (single-tape, single-head) Turing Machine. Any computation that can be performed by one of these models can also be simulated by a standard Turing Machine, although with possibly greater computational overhead.

Typically, a Turing Machine designed to compute one algorithm may fail or be inefficient at carrying out another algorithm. In general, each algorithm has a specific Turing Machine that is best suited, or most efficient, for executing it.

This raises the question of whether there exists a single Turing Machine capable of executing every algorithm, given an appropriate encoding of the algorithm. Perhaps the most remarkable answer is provided by the existence of such a machine: the **Universal Turing Machine**. The Universal Turing Machine laid the foundation for the modern concept of general-purpose computers, where a single device can perform diverse tasks by simply changing its software. Turing's idea of universality is a cornerstone of theoretical computer science. A Universal Turing Machine operates by reading the description of a specific Turing Machine and its input from its tape and then executing the encoded instructions.

Definition 2.4. *A **Universal Turing Machine** is a Turing Machine U that takes as input $\langle M, w \rangle$, where M is another Turing Machine and w is an input string, and emulates the computation of M on input w, i.e., for any input $\langle M, w \rangle$, the machine U computes $M(w)$.*

All Turing Machines share a standard structure, which can itself be regarded as a set of rules—for example, the transition function, which has the same structural form on every machine. Given this uniformity, it is conceivable to construct a Universal Turing Machine that can simulate any other Turing Machine.

Proposition 2.2. *There exist a Universal Turing Machine.*

Partial recursive functions are a fundamental concept in the theory of computation, representing a class of functions that are computable using algorithms. These functions may not produce an output for every input, hence the term "partial," as opposed to "**total**" functions, which are defined for all inputs. They play a critical role in the study of Turing Machines by formalizing the notion of algorithmic processes. Every partial recursive function corresponds to a computation that can be performed by some Turing Machine.

Definition 2.5. *From [B3]: A k-place **partial recursive function**, without loss of generality, is a function $f : \mathbb{N}^k \to \mathbb{N}$ that is computable by a Turing Machine. f is **total** if its domain is all of $\mathbb{N}^k$.*

In other words, a recursive function is one that can be defined by a finite sequence of operations, each of which can be performed by a Turing Machine. This class of functions is fundamental to the theory of computation, as it encapsulates the idea of functions that can be computed algorithmically. Recursive functions form the basis for defining computability, as every function that can be computed by an algorithm is a recursive function, and thus, can be modeled by a Turing Machine.

A more convenient notation is required to describe Turing Machine computations. Previously, M was used to denote Turing Machines in general. Since Turing Machines represent computations of partial recursive functions, $M(X)$ is now used to denote a Turing Machine that takes an input X.

X may simply refer to a location on the tape which contain $\{X\}$. When $M(X)$ is executed on a particular value of X, it is denoted by $M(\{X\})$, where $\{X\}$ represents the current value of X on the tape.

The entire tape of the Turing Machine can be encoded as

$$\langle M(*X), X = \texttt{value} \rangle,$$

which evolves during computation to

$$\langle M(*\texttt{value}) \rangle$$

and eventually halts in

$$\langle \{M(\texttt{value})\}* \rangle.$$

The head position on the tape is explicitly marked with $*$ to indicate the symbol currently being read or written.

In a more compact form, the initial configuration can be written as $\langle M(*X) \rangle$, which evolves to $\langle M(*\{X\}) \rangle$ and finally halts at $\langle \{M(\{X\})\}* \rangle$. In this way, the need to explicitly designate the values of variables

(X) is eliminated, as they are instead implied by the notation. The Universal Turing Machine U, on which $\langle M(X) \rangle$ is executed, can initially replace X in $\langle M(X) \rangle$ with $\{X\}$, although such a replacement is not strictly necessary.

Proposition 2.3. *From [B3]: There exists only a countably infinite number of partial recursive functions, denoted by $\aleph_0$, as well as a countably infinite number of recursive functions, also represented by $\aleph_0$.*

Each partial recursive function can be associated with an enumeration φ_e for some integer e. Specifically, given a Universal Turing Machine U, if $\langle M_{\varphi_e}(X) \rangle$ is the input encoding of the Turing Machine that computes φ_e (accepting and returning $\varphi_e(x)$ for inputs x where φ_e is defined, and does not halt otherwise), then e can be identified as $e = \langle M_{\varphi_e}(X) \rangle$. Or,

$$e = \langle M_e(X) \rangle \equiv \langle \texttt{COMPUTE } \varphi_e(X) \rangle$$

more conveniently, which define a mapping from Turing Machines $\mathcal{M}$ to partial functions $\mathcal{F}$, and establishes a correspondence between Turing programs and the partial functions they compute via a universal machine.

Since every partial recursive function has a corresponding Turing Machine, there exists a **universal partial recursive function** that corresponds to the Turing Machine capable of computing all partial recursive functions.
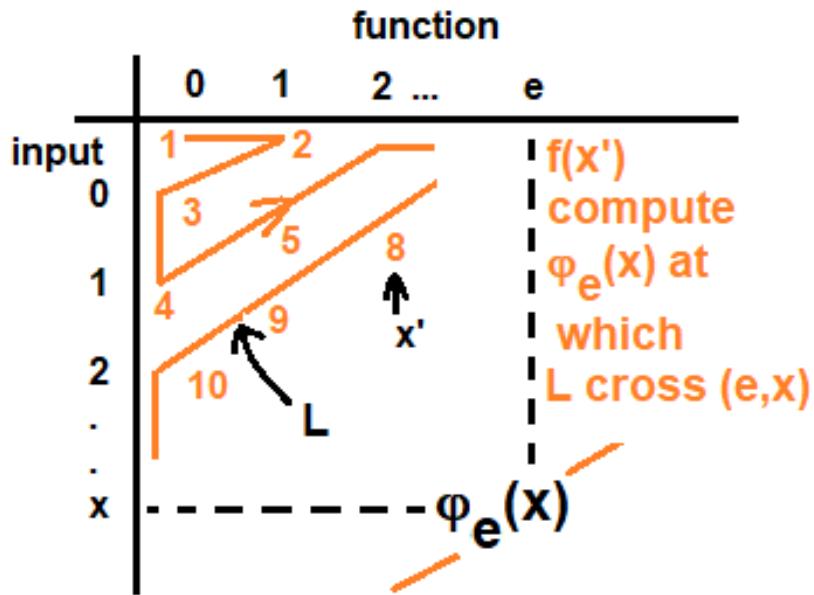

Figure 7: Example constructing a universal partial recursive function: *The construction of a universal partial recursive function can be represented by a computational grid. Let the rows correspond to indices e of partial recursive functions φ_e, and the columns correspond to input values x. Each point (e, x) in the grid is labeled with the value $\varphi_e(x)$, if defined. A **computational line** L can then be drawn to connect the points sequentially according to a chosen enumeration of the function-input pairs, representing the step-by-step computation of the universal function that systematically produces the outputs of all φ_e. Each visit to a e, x by L is enumerated by x'. The computational line L must pass through each point (e, x) in the grid, but it is not required that each point be visited only once; the essential requirement is that the line is constructed according to a fixed arithmetic procedure which ensure a computational relationship between x' and (e, x).*

Self-referentiality arises in mathematics when an expression or construction is capable of referring to itself within its own formal system. Such self-reference becomes possible when mathematical objects, such as functions or computational procedures, can be represented by numerical encodings. Once an

object can access or manipulate its own encoding, it can construct statements or computations that effectively describe or act upon itself. This idea forms the foundation for many fundamental results in logic and computability, including Gödel's incompleteness theorem and Kleene's recursion theorem. In the context of computation, self-referentiality allows a Turing Machine to simulate the effect of "feeding its own description" as input, thereby formalizing the concept of a system that reasons about or operates on its own structure.

To formalize the notion of **self-referentiality** in Turing Machines, consider a variable denoted by SELF. A Turing Machine M may take two inputs, SELF and X, written as $M(\text{SELF}, X)$. Here, {SELF} represents the encoding of the machine itself, that is,

$$\{\text{SELF}\} = \langle M(\text{SELF}, X)\rangle.$$

When the configuration $\langle M(*\text{SELF}, X)\rangle$ is given as input to a Universal Turing Machine U, the universal machine can perform the substitution

$$\text{SELF} \mapsto \langle M(\text{SELF}, X)\rangle,$$

so that the computation proceeds through the configuration

$$\langle *M(\{\text{SELF}\}, \{X\})\rangle,$$

and ultimately halts in

$$\langle \{M(\{\text{SELF}\}, \{X\})\}*\rangle.$$

This construction represents a Turing Machine that can operate on its own description, thereby exhibiting explicit self-reference. The variable SELF acts as a placeholder for the machine's own encoding, enabling the formal analysis of self-referential computations and fixed points.

Proposition 2.4. *Fixed point theorem: For any total recursive function $f : \mathbb{N} \to \mathbb{N}$, there exists an integer e such that $\varphi_e(.) = \varphi_{f(e)}(.)$.*

Proof (intuitive construction). : Consider a Universal Turing Machine U capable of running any encoded computation of the form $\langle M_{f(\text{SELF})}(X)\rangle$, where

$$\{\text{SELF}\} = \langle M_{f(\text{SELF})}(X)\rangle.$$

Since the encoding of the machine is equal to its own self-reference, that is, $\langle M_{\text{SELF}}(X)\rangle = \{\text{SELF}\}$, the Turing Machine

$$M_{f(\{\text{SELF}\})}(X)$$

evaluates the same result as

$$M_{\{\text{SELF}\}}(X).$$

Hence, for a given total recursive function f, there exists an index

$$e = \langle M_{f(\text{SELF})}(X)\rangle$$

such that

$$\varphi_e(X) = \varphi_{f(e)}(X).$$

This establishes the existence of a fixed point e satisfying the fixed point property. $\square$

$M_{f(e)}(X)$ and $M_{f(\text{SELF})}(X)$ are entirely different Turing machines; the latter requires self-replication.

As the name suggests, the Fixed Point Theorem asserts that for every total recursive function f, there exists an index e such that the transformation $\varphi_{f(e)}$ remains invariant in behavior to φ_e, that is, $\varphi_{f(e)}(X) = \varphi_e(X)$..

However, there is an even deeper implication: if a universal Turing Machine U applies a total recursive transformation f to an input description $e = \langle M_e(X)\rangle$, thereby producing a new program $M_{f(e)}(X)$ that performs a more complex operation, then there exists a Turing Machine $M_{e'}(X)$ that inherently carries out this transformation upon itself. In other words, $M_{e'}$ is capable of generating $M_{f(e')}(X)$ autonomously, without requiring the external application of f by U.

Proposition 2.5. *For any total recursive transformation f, there exists a Turing Machine M_e that internally realizes its own transformation, such that $M_e(X)$ computes identically to $M_{f(e)}(X)$ without requiring the external application of f.*

For instance, the function f may take the description $e = \langle M_e(X) \rangle$ and simply produce two copies of it, $\langle M_e(X), M_e(X) \rangle$. However, the Fixed Point Theorem implies the existence of a self-replicating Turing Machine $M_r(X)$ that generates its own description autonomously. Previously, it was assumed that the variable SELF was managed by the Universal Turing Machine used to execute $\langle M_e(\text{SELF}) \rangle$; yet it is now asserted that there exists a Turing Machine $M_s(\text{SELF})$ capable of performing this substitution internally, without external intervention. These are particularly interesting examples illustrating the applications of the Fixed Point Theorem.

There exists a Turing machine $\text{HIST}(X, n)$ that, given the description $\langle M \rangle$ of any Turing machine M, outputs an encoded computational history of M for n time steps. The output contains, in an appropriate encoding, the head position, tape cell values, and internal state at each step—effectively forming a log file of the computation up to time n.

The function computed by $\text{HIST}(X, n)$ is total and recursive, since for any finite number of steps n, the behavior of M can be deterministically simulated for exactly n transitions. Therefore, by the Fixed Point Theorem, there exists a Turing machine M_h that outputs an encoding of its own computational log to n time steps, effectively generating a finite description of its own execution history.

$$\langle M_h \rangle = \{\text{SELF}\} = \langle \text{LOOP}(\text{SELF}, n) \rangle$$

Note: $HIST(\text{SELF}, n)$ is a completely different program from $HIST(X, n)$ in U, and not a mere substitution of $\{X\}$ by $\{\text{SELF}\}$.

The Turing machine $HIST(X, n)$ takes as input a description $\langle M \rangle$ of a Turing machine M and produces a description of its computation up to n runs. In this sense, $HIST(X, n)$ represents a partial recursive function that outputs an encoding of the behavior of M over its execution.

What Turing Machines are capable of is indeed fascinating. Equally intriguing, however, are the tasks that they cannot perform, highlighting fundamental limits of computation.

Proposition 2.6. *From [B3] There are $2^{\aleph_0}$ functions of the form $f : \mathbb{N}^k \to \mathbb{N}$, implying that some functions cannot be recursive or computable.*

Definition 2.6. *The set H is called the **Halting Set for Turing Machines**. It represents the set of all encoded pairs $\langle M(X), w \rangle$, where $M(X)$ is a Turing Machine and $M(X)$ halts on the input w.*

$$H = \{\langle M(X), w \rangle \mid M(X) \text{ is a Turing Machine and } M(X) \text{ halts on } w\}.$$

Proposition 2.7. *H is undecidable.*

Proof. Let D be the Turing Machine that decides membership in H. Then the corresponding Turing Machine to the following is valid:

$$\langle M \rangle = \left\langle \begin{cases} \text{if } D(\text{SELF}) \text{ ACCEPT, then LOOP.} \\ \text{else HALT.} \end{cases} \right\rangle$$

When M is run a contradiction arises. If $D(\text{SELF})$ accepts, then M must loop. Conversely, if $D(\langle \text{SELF} \rangle)$ rejects, then M halts, contradicting the result of D. Therefore, either D cannot be consistent, or D is not a Turing Machine, in which case, nor is M a Turing Machine which access D.

$\square$

In the context of a Turing Machine, self-referencing occurs when the machine executes a sequence of operations that lead it to process or manipulate its own description or encoding. A Turing Machine $M_e(X)$ is defined by its encoding $e = \langle M_e(X) \rangle$, which represents the machine's structure and rules. $M_e(X)$ may process its own description e. This can happen directly, where $M_e(X)$ parses e explicitly, or indirectly, such as when a Turing Machine $M_a(X)$ processes $b = \langle M_b(X) \rangle$, which in turn processes $a = \langle M_a(X) \rangle$, thereby inducing self-referencing through the computational path.

The **general undecidability problem** in Turing Machines refers to the inherent limitation that no single Turing decider D_Φ, can universally determine whether a property ϕ holds for all possible Turing Machine descriptions.

Let,

$$\Phi = \{ \langle M, w \rangle \mid M \text{ is a Turing Machine and } M \text{ satisfy } \phi \}.$$

Proposition 2.8 (Rice's Theorem). Φ *is undecidable*

Proof. Construct:

$$\langle M \rangle = \left\langle \begin{cases} \text{if } D_\phi(\text{SELF}) \text{ ACCEPT, then } \neg\phi \\ \text{else } \phi \end{cases} \right\rangle$$

□

When Turing machines take descriptions of other Turing machines as input to decide certain properties, deciders can be classified based on how much of the input machine's computation they simulate. A **full-run decider** simulates the input machine completely, running it until it halts (if it ever does) to determine the property. Such deciders are fundamentally limited: if the input machine does not halt, the decider cannot produce an output. More generally, an n-**run decider** simulates the input machine for at most n runs (or steps), producing a decision within this bounded computation. This notion captures a spectrum of deciders, from fully simulating to partially analyzing the input machine, highlighting the interplay between computability and the behavior of machines operating on descriptions of other machines.

Definition 2.7. *A Turing machine $D_n(Y)$ is called an n-**run decider** if, given the input description $Y = \langle M, w \rangle$ as input, it decides by running $M(w)$ for at most n runs (or steps).*

Definition 2.8. *The set H_n is called the **n-Halting Set for Turing Machines**. It represents the set of all encoded pairs $\langle M(X), w \rangle$, where $M(X)$ is a Turing Machine and $M(X)$ halts on the input w within n runs.*

$$H_n = \{ \langle M(X), w \rangle \mid M(X) \text{ is a Turing Machine and } M(X) \text{ halts on } w \text{ within } n\text{-runs} \}.$$

Proposition 2.9. *H_n is decidable, by a n-run decider.*

Proof. If a decider $D_{n'}(Y)$, with $n' < n$ exist, it is possible to construct:

$$\langle M \rangle = \left\langle \begin{cases} \text{if } D_{n'}(\text{SELF}) \text{ ACCEPT, then run LOOP.} \\ \text{else HALT now at } n' < n. \end{cases} \right\rangle$$

which result in contradiction proving that D'_n is either inconsistent or not a Turing Machine.

This construction ensures that any prediction made by $D_{n'}$ about halting within n runs can be contradicted by M's subsequent behavior. After simulating $D_{n'}$, the computation of M is still at step $n' < n$. Therefore, M can either continue running to reach n steps or halt immediately. If $D_{n'}$ predicts halting,

M continues running to n steps; if $D_{n'}$ predicts no halting, M halts early at $n' < n$. Consequently, $D_{n'}$ cannot correctly decide membership in H_n.

Thus, a decider must simulate all n runs to determine whether a machine halts within n steps. Any decider limited to fewer than n runs is insufficient. □

An identical argument applies to the general n-undecidability problem Φ_n, establishing a deeper result about computation: a universal decider cannot determine membership in Φ_n by partially simulating the input Turing machine. In other words, any decider attempting to resolve Φ_n must, in principle, execute all n steps of the input machine; partial execution is insufficient.

Both in the proof of the undecidability of the Halting Problem and in the general undecidability theorems, it is established that no Turing machine can serve as a decider D for these problems. It is a fundamental truth that every Turing machine either halts or loops on a given input; however, there exists no algorithmic procedure that can determine this behavior for an arbitrary Turing machine. This does not however preclude the existence of hypothetical machines whose computational power exceeds that of standard Turing machines, and can decide H. Such machines whose computational power exceed beyond Turing Machines are called **oracles**. An oracle is a conceptual entity that provides the output of a function instantaneously, without internal computation, analogous to a lookup table with immediate access.

When a Turing machine M is given access to an oracle O, it is referred to as an **oracle Turing machine**, denoted M^O. The oracle thereby defines a new class of computational power corresponding to the set of computations accessible to machines equipped with it. Nevertheless, even within this class, particularly when M^O has access to the oracle O_H that decides the Halting Problem for ordinary Turing machines, a new Halting Problem arises that is undecidable within the class of M^{O_H} machines. Solving this new problem requires an even more powerful oracle. This iterative process gives rise to a **hierarchy of computational power**, often called the **arithmetical hierarchy**, where each level corresponds to machines with access to increasingly powerful oracles.

Definition 2.9. *let the O_H be the oracle that decide the halting set H. let the class of all Oracle Turing Machines with access to O_H be TM^{O_H}. The halting set H' for the class of Oracle Turing Machines $TM^{O_H}(X)$, represents the set of all encoded pairs $\langle M^{O_H}(X), w \rangle$, where $M^{O_H}(X)$ halts on the input w.*

$$H' = \{ \langle M^{O_H}(X), w \rangle \mid M^{O_H}(X) \in TM^{O_H} \text{ , and halts on } w \}.$$

Proposition 2.10. *H' is undecidable.*

Proof. Let $D' \in TM^{O_H}$ be the Turing Machine that decides membership in H'. Then the corresponding Oracle Turing Machine to the following is valid ($M' \in TM^{O_H}$):

$$\langle M' \rangle = \left\langle \begin{cases} \text{if } D'(\text{SELF}) \text{ ACCEPT, then LOOP.} \\ \text{else HALT.} \end{cases} \right\rangle$$

When M is run a contradiction arises. Therefore D' is inconsistent, or is in a higher class than TM^{O_H}.

□

So far, models of Turing Machines have been assumed to be deterministic. There exists, however, a nondeterministic version of Turing Machines that is useful in capturing nondeterministic aspects of computations. These Turing Machines are called **Nondeterministic Turing Machines** (NTM) In this model, the machine may have multiple possible transitions from a given configuration, and the computation can proceed along any of these possible paths. Conceptually, this allows the machine to explore many computational branches simultaneously, a property that gives rise to the theoretical foundation of nondeterministic computation and the complexity class NP. Nondeterminism serves as

an important abstraction in understanding what could be computed efficiently if parallel exploration of possibilities were possible.

Definition 2.10. *A **Nondeterministic Turing Machine** (NTM) is similar to a deterministic Turing machine except that the transition function is replaced by a transition relation*

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

where $\mathcal{P}(S)$ denotes the power set of S. At each computation step, the machine may choose any valid transition among those specified by δ. An NTM is said to accept an input string if there exists at least one computation path that leads to a halting state in F.

All halting branches of a nondeterministic Turing machine have finite depth. Therefore, a nondeterministic Turing machine can have at most a countably infinite number of halting outputs, corresponding to the cardinality of $\mathbb{N}$.

A nondeterministic Turing machine may possibly have infinitely many halting branches. There exists a systematic computational method to enumerate all halting outputs, provided the machine is finitely branching. For example, by performing a **breadth-first exploration** of the computation tree, every finite halting branch will eventually be reached and its output recorded. This method, often referred to as **dovetailing**, guarantees that each halting output can be assigned a unique index in a sequence, producing a countable enumeration of all outputs. Consequently, the halting outputs of any finitely branching Nondeterministic Turing Machine can be represented as a partial recursive function $\varphi_e(x, n)$, where n indexes the halting branch in the enumeration, and e an encoding of the description of the nondeterministic Turing Machine that computes it.

Proposition 2.11. *Nondeterministic and deterministic Turing machines are equivalent in computational power.*

The **Boolean Satisfiability Problem (SAT)** is defined as follows: given a Boolean formula ϕ composed of variables, logical connectives (AND, OR, NOT), and possibly parentheses, determine whether there exists an assignment of truth values to the variables that makes ϕ evaluate to TRUE. If such an assignment exists, the formula is said to be *satisfiable.*

Definition 2.11 (SAT). *Given a Boolean formula $\phi(x_1, x_2, \ldots, x_n)$, the **SAT problem** asks whether there exists a vector $(b_1, b_2, \ldots, b_n) \in \{0, 1\}^n$ such that*

$$\phi(b_1, b_2, \ldots, b_n) = 1.$$

SAT is an important problem in computational complexity because it belongs to the class **NP**. A problem is in NP if a non-deterministic Turing machine can decide it in polynomial time. For SAT, a non-deterministic Turing machine can guess an assignment of variables and verify whether it satisfies the formula in time polynomial in the number of variables and clauses.

Proposition 2.12. *SAT $\in$ NP.*

Proof. An n-variable Boolean formula ϕ has 2^n possible truth assignments. A deterministic Turing machine would, in the worst case, need to test all of them sequentially, taking exponential time. However, a nondeterministic Turing machine can exploit its branching nature to explore all 2^n configurations simultaneously. Each branch corresponds to one assignment, and the machine can verify whether ϕ evaluates to TRUE along that branch in time polynomial in n. Hence, the entire computation runs in $O(n)$ nondeterministic time, proving that SAT is in NP. $\qquad\square$

Proposition 2.13 (Boolean Encoding Theorem). *For any Turing machine M, input w, and time bound N, there exists a Boolean formula $\phi_{M,w,N}$ of size polynomial in N such that*

$$\phi_{M,w,N} \text{ is satisfiable} \iff M \text{ accepts } w \text{ within } N \text{ steps.}$$

Since, in N steps, the head cannot move more than N cells, and whether a Turing machine M accepts within N steps is a finite decision problem, it defines a Boolean function on $O(N)$ bits encoding the computation history. The key insight in the proof, is that the validity of such a computation can be checked using only local consistency conditions between adjacent time steps and neighboring tape cells. Each local condition can be expressed as a constant-size Boolean constraint, and the total number of constraints is polynomial in N.

The class NP includes all decision problems whose solutions can be verified by a Nondeterministic Turing Machine in polynomial time. However, within NP, there exist certain problems that are as hard as any other problem in the class. These are known as **NP-complete problems**.

A **decision problem** can be represented as a language $L \subseteq \Sigma^*$, where Σ is a finite alphabet and each string $x \in \Sigma^*$ represents an instance of the problem. A string $x \in L$ indicates that the answer to the problem instance encoded by x is "yes," while $x \notin L$ indicates a "no" instance.

Definition 2.12. *A decision problem (language) L is said to be **NP-complete** if:*

1. *$L \in NP$, that is, L can be verified in polynomial time by a nondeterministic Turing machine.*

2. *Every problem $L' \in NP$ can be reduced to L in polynomial time; formally, there exists a polynomial-time computable function f such that:*

$$x \in L' \iff f(x) \in L.$$

Proposition 2.14 (Cook-Levin Theorem)**.** *The Boolean satisfiability problem (SAT) is NP-complete.*

Proof. If there exists a decider program that operates in time $O(n^k)$, then by the Boolean Encoding Theorem, its computation can be represented as a Boolean formula of polynomial size whose satisfiability corresponds to the decider accepting the input.

$\square$

The significance of SAT being **NP-complete** lies in its central role within the study of computational complexity. Since SAT belongs to the class **NP**, any candidate solution can be **verified in polynomial time** by a deterministic Turing machine. However, finding a satisfying assignment, if one exists, may require exploring an **exponentially large space** of possible assignments in the worst case.

The **P vs NP problem** asks whether there exists a **deterministic polynomial-time algorithm** capable of solving SAT, and consequently, all problems in NP. Currently, no such algorithm is known, and it is widely conjectured that $\mathbf{P} \neq \mathbf{NP}$, meaning that while solutions can be verified quickly, they cannot necessarily be found efficiently.

Definition 2.13 (Asymptotic Growth Notations)**.** *Let $f(n)$ and $g(n)$ be functions from $\mathbb{N}$ to $\mathbb{R}^+$. The following asymptotic notations are defined as follows:*

- ***Big-O:*** *$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,*

$$f(n) \leq c \cdot g(n).$$

- ***Big-Theta:*** *$f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,*

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

- ***Big-Omega:*** *$f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,*

$$f(n) \geq c \cdot g(n).$$

- ***Little-o:*** *$f(n) \in o(g(n))$ if*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

- **Little-omega:** $f(n) \in \omega(g(n))$ if
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty.$$

The following propositions present fundamental results in computational complexity theory, illustrating key relationships between time and space resources for deterministic and nondeterministic Turing machines, as well as the existence of hierarchies and trade-offs in computational resources.

Proposition 2.15 (Savitch's Theorem). *Any problem solvable by a nondeterministic Turing machine (NTM) using $s(n)$ space can be solved by a deterministic Turing machine (DTM) using at most $s(n)^2$ space.*

Proposition 2.16 (Time Hierarchy Theorem). *Let $t_1(n)$ and $t_2(n)$ be time-constructible functions such that $t_2(n) \in \omega(t_1(n) \log t_1(n))$. Then there exists a language L such that*
$$L \in TIME(t_2(n)) \quad but \quad L \notin TIME(t_1(n)).$$

Proposition 2.17 (Space Hierarchy Theorem). *Let $s_1(n)$ and $s_2(n)$ be space-constructible functions such that $s_2(n) \in \omega(s_1(n))$. Then there exists a language L such that*
$$L \in SPACE(s_2(n)) \quad but \quad L \notin SPACE(s_1(n)).$$

Proposition 2.18 (Time–Space tradeoff for deterministic Turing machines). *Let M be a deterministic Turing machine that decides a language L using time $t(n)$ and space $s(n)$. Then for any $f(n)$ satisfying $s(n) \le f(n) \le t(n)$, there exists a simulation of M using space $f(n)$ and time*
$$O\Big(\frac{t(n)^2}{f(n)}\Big).$$

Beyond Turing Machines

There are undecidable problems that raise seemingly reasonable questions. For example, the Halting Problem asks whether a given Turing Machine halts on a given input. This question seems reasonable, as any Turing Machine either halts or loops indefinitely on its input.

The self-contradictory Turing Machine for a decider D previously constructed:

$$\langle X \rangle = \langle \text{do} : \neg D(\langle X \rangle) \rangle$$

can be interpreted in the following manner. D exists, but it is not a Turing Machine. Therefore, X cannot be a Turing Machine if it refers to D. Mathematicians have introduced **oracle machines** to represent those machines that cannot be explicitly constructed but are instead given as black-box entities. Oracles can be thought of as simple lookup tables, which specify input-output pairs without any internal structure or mechanism, So that inputs can be given, and outputs can be obtained.

Definition 2.14. *An **oracle machine** O_f for a given function f, return $f(x)$ for input x, and contains no internal mechanisms or structures.*

Definition 2.15. *An **oracle Turing Machine** T^{O_f} has access to an oracle O_f and a Turing Machine T.*

There is no oracle Turing Machine that can compute all functions. Instead, oracle Turing Machines form a hierarchical ladder of computational power.

Definition 2.16. *Let $A, B \subseteq \mathbb{N}$ (sets of natural numbers representing decision problems). A is **Turing reducible** to B, written*
$$A \le_{\mathcal{T}} B,$$
if there exists an oracle Turing Machine T^{O_B} such that O_B decides membership in A. In other words, A can be decided using O_B as an oracle.

Definition 2.17. *The **Turing degree** of a set $A \subseteq \mathbb{N}$ is the equivalence class*

$$\mathcal{T}(A) = \{B \subseteq \mathbb{N} : A \equiv_{\mathcal{T}} B\},$$

where $A \equiv_{\mathcal{T}} B$ means both $A \leq_{\mathcal{T}} B$ and $B \leq_{\mathcal{T}} A$. Two sets have the same Turing degree if they can compute each other.

Definition 2.18. *The collection of Turing degrees, ordered by $\leq_{\mathcal{T}}$, forms the structure known as the **degrees of unsolvability**. At the bottom is the degree of decidable sets (denoted 0), and strictly above it lies the degree of the Halting problem (denoted $0'$).*

Definition 2.19. *Given a set $A \subseteq \mathbb{N}$, the **Turing jump** of A, written A', is defined as*

$$A' = \{\langle M, w \rangle : M \text{ run on } T^{O_A} \text{ HALTS on } w\},$$

Proposition 2.19. *For every $A \subseteq \mathbb{N}$, there exist $A <_{\mathcal{T}} A'$. That is, the Turing jump strictly increases computational power.*

Iterating the jump yields a hierarchy of degrees:

$$0 < 0' < 0'' < 0^{(3)} < \cdots,$$

which corresponds to the levels of the **arithmetical hierarchy**. The diagonal argument guarantees that no oracle, even one solving the halting problem for A, can decide A' itself.

Read more on [B7]
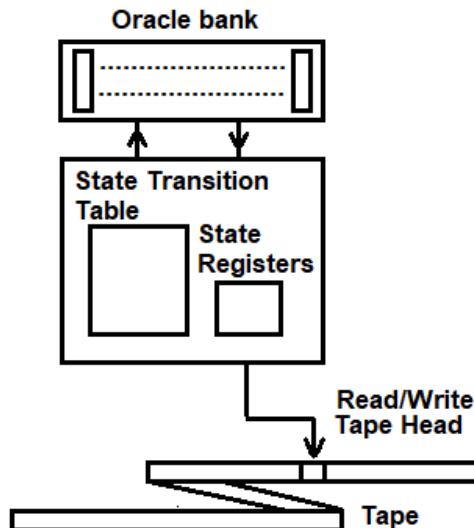
Figure 8: Oracle Turing Machine: *A schematic representation of an Oracle Turing Machine, featuring the standard components of a Turing Machine, including the state registers, transition table, tape, and read/write head, along with a bank of oracles. Each oracle provides access to a different non-computable function, allowing the machine to query multiple oracles depending on the problem at hand. This extended architecture models computational systems that operate beyond classical Turing limits, enabling analysis of decision problems in higher complexity classes.*

3   Algorithmic Information Theory

Kolmogorov complexity provides a way to quantify the amount of information in a string by measuring the length of its shortest description in a fixed computational model.

Definition 3.1. *The **Plain Kolmogorov Complexity** of a string x, denoted $C_U(x)$, with respect to a Universal Turing Machine U, is defined as the length of the shortest possible program p for U that outputs x when run on an empty input tape. Formally, it is:*

$$C_U(x) = \min_p \{|p| \mid U(p) = x\}$$

where $|p|$ is the length of the program p, and $U(p) = x$ means that the universal Turing Machine U halts and outputs the string x when given p as input.

In other words, $C_U(x)$ represents the smallest number of bits required to describe the string x in the form of a program for a universal Turing Machine U, essentially quantifying the complexity of x in terms of the computational resources needed to generate it.

Proposition 3.1. *Most strings are in-compressible*

Proof. The probability of a string being incompressible within $n - c$ is given by:

$$p = 1 - \frac{2^{(n-c+1)} - 1}{2^n} \approx 1$$

$\square$

A **prefix-free code** is a type of encoding scheme in which no code word is a prefix of any other code word. In other words, a prefix-free code is a set of strings such that for any two distinct strings x and y in the set, x is not a prefix of y and vice versa. Formally, a set of code words $\{c_1, c_2, \ldots, c_n\}$ is a prefix-free code if for any pair of code words c_i and c_j, with $i \neq j$, the condition:

$$\forall i \neq j \quad (c_i \npreceq c_j \text{ and } c_j \npreceq c_i)$$

holds, where $\preceq$ denotes the prefix relation. That is, no code word c_i is a prefix of any other code word c_j. prefix-free codes are crucial in information theory, particularly in data compression and error detection, as they ensure that encoded messages can be uniquely decoded without ambiguity.

Proposition 3.2. *From [B6]: Let $\{c_1, c_2, \ldots, c_n\}$ be a set of code words for a prefix-free code, and let $|c_i|$ denote the length of code word c_i. The set of code words satisfies **Kraft's inequality** given by:*

$$\sum_{i=1}^{n} 2^{-|c_i|} \leq 1$$

where $|c_i|$ is the length of the i-th code word, and the sum runs over all code words in the set.

This inequality is necessary and sufficient for the existence of a prefix-free code. If the inequality holds, there exists a prefix-free encoding for the set of code words. Conversely, if the inequality does not hold, no such prefix-free code exists.

To turn any binary string s into a prefix-free code, construct the new string by first taking a string of $0^{|s|}$, followed by a single 1, and then appending the original string s. This construction ensures that the

resulting string is prefix-free, as no code word is a prefix of another. Specifically, for a given string s, the prefix-free code is formed as $0^{|s|}1s$, where $|s|$ represents the length of the string s.

A **prefix-free machine** is a Turing machine U whose set of valid programs forms a prefix-free set: no valid program is a prefix of another. Formally, a set of binary strings P is **prefix-free** if for any distinct $p, q \in P$, p is not a prefix of q. Prefix machines are useful because they allow probabilistic interpretations and satisfy Kraft's inequality.

Definition 3.2 (Prefix-Free Kolmogorov Complexity). *Let U be a fixed universal prefix machine, i.e., a Turing machine whose set of valid programs forms a prefix-free set. The prefix-free Kolmogorov complexity of a string $x \in \{0,1\}^*$ is*

$$K_U(x) = \min\{ |p| \mid U(p) = x \},$$

where the minimum is taken over all programs p that produce x and are valid for U.

Proposition 3.3 (Generation of prefix-free codes). *Let $\{\ell_i\}_{i\in\mathbb{N}}$ be a computably enumerable sequence of positive integers satisfying Kraft's inequality*

$$\sum_i 2^{-\ell_i} \leq 1.$$

Then there exists a computably enumerable prefix-free set of binary strings $\{p_i\}_{i\in\mathbb{N}}$ such that $|p_i| = \ell_i$ for all i.

Proof. For each length ℓ_i, recursively choose the next available rightmost path in the binary tree of codewords. By construction, no codeword is a prefix of another, so the resulting set is prefix-free. This method is known as the **greedy algorithm**. $\square$

Proposition 3.4 (Prefix-Free Simulation of Turing Machines). *Let M be any Turing machine. Then there exists a prefix-free machine M' that simulates M, in the sense that for every input $x \in \{0,1\}^*$,*

$$M'(p_x) = M(x),$$

where p_x is a prefix-free code corresponding to x.

Proof. Let $\{s_i\}_{i\in\mathbb{N}}$ be a computably enumerable prefix-free code, and let $\{x_i\}_{i\in\mathbb{N}}$ be an enumeration of all binary strings $\{0,1\}^*$.

Construct the prefix-free machine M' as follows:

1. On input p, M' enumerates the prefix-free codewords $s_1, s_2, \ldots$ in order, until it finds i such that $s_i = p$.

2. Once the matching index i is determined, M' retrieves the corresponding string x_i from the enumeration of $\{0,1\}^*$.

3. M' simulates M on input x_i and outputs the result.

$\square$

Proposition 3.5 (Invariance Theorem). *Let U and V be two universal prefix machines. Then there exists a constant $c_{U,V}$, depending only on U and V, such that for all strings $x \in \{0,1\}^*$,*

$$|K_U(x) - K_V(x)| \leq c_{U,V}.$$

Proof. Since U is universal, there exists a fixed program $p_{V \to U}$ that allows U to simulate V on any input. Hence, for every string x, if q is a shortest program for V producing x, then U can produce x by first running $p_{V \to U}$ to simulate V, and then feeding it q. Therefore,

$$K_U(x) \leq K_V(x) + |p_{V \to U}|.$$

Similarly, there exists a constant $|p_{U \to V}|$ for simulating U on V, giving

$$K_V(x) \le K_U(x) + |p_{U \to V}|.$$

Combining these inequalities,

$$|K_U(x) - K_V(x)| \le c_{U,V}, \quad \text{where } c_{U,V} = \max\{|p_{V \to U}|, |p_{U \to V}|\}.$$

$\square$

Definition 3.3 (Lower semicomputable semimeasure). *Let $\Sigma = \{0,1\}$ and Σ^* denote the set of finite binary strings. A function*

$$\mu : \Sigma^* \to [0,1]$$

is called a semimeasure *if*

$$\sum_{x \in \Sigma^*} \mu(x) \le 1.$$

It is called lower semicomputable *if there exists a total computable function*

$$\mu(x,t) : \Sigma^* \times \mathbb{N} \to \mathbb{Q}$$

such that for all x:

1. *$\mu(x,t) \le \mu(x,t+1)$ for all t,*

2. *$\lim_{t \to \infty} \mu(x,t) = \mu(x)$.*

Proposition 3.6 (Dropping normalization yields a lower semicomputable semimeasure). *Let $f_a : \Sigma^* \to [0,\infty)$ be a computable function such that*

$$Z(a) := \sum_{x \in \Sigma^*} f_a(x) < \infty.$$

Define

$$P(x) := \frac{f_a(x)}{Z(a)}.$$

Then there exists a constant $c > 0$ such that the function

$$\mu(x) := c\, f_a(x)$$

is a lower semicomputable semimeasure. Moreover, μ multiplicatively dominates P, i.e.

$$P(x) \le \frac{1}{cZ(a)}\, \mu(x) \quad \text{for all } x \in \Sigma^*.$$

Proof. Since f_a is computable, it is both upper and lower semicomputable. Choose a constant $c > 0$ such that $cZ(a) \le 1$ (for example, $c = 2^{-k}$ for any k with $Z(a) \le 2^k$). Then

$$\sum_{x \in \Sigma^*} \mu(x) = c \sum_{x \in \Sigma^*} f_a(x) = cZ(a) \le 1,$$

so μ is a semimeasure.

Lower semicomputability of μ follows from that of f_a, since multiplication by a rational constant preserves lower semicomputability. Finally, for all x,

$$P(x) = \frac{f_a(x)}{Z(a)} \le \frac{1}{cZ(a)}\, \mu(x),$$

establishing multiplicative dominance. $\square$

Proposition 3.7 (Uniform computable approximation on compact sets). *Let $K \subset \mathbb{R}^n$ be compact, and let*

$$f : K \to \mathbb{R}$$

be a computable function in the sense of computable analysis. Suppose that f admits a computable modulus of continuity, i.e. there exists a computable function

$$\omega : \mathbb{N} \to \mathbb{N}$$

such that for all $x, y \in K$ and all $n \in \mathbb{N}$,

$$\|x - y\| < 2^{-\omega(n)} \implies |f(x) - f(y)| < 2^{-n}.$$

Then there exists a computable sequence of functions

$$(f_n)_{n \in \mathbb{N}}$$

with each $f_n : K \to \mathbb{R}$ computable, such that

$$\sup_{x \in K} |f_n(x) - f(x)| \le 2^{-n} \quad \text{for all } n \in \mathbb{N}.$$

In particular, f can be uniformly approximated on K by computable functions to arbitrary precision.

Can Randomness beat complexity?

Proposition 3.8 (Upper bound on Kolmogorov complexity by a computable distribution). *Let $P : \Sigma^* \to [0, 1]$ be a computable probability distribution on finite binary strings, i.e. there exists a computable function that outputs $P(x)$ to arbitrary precision for each $x \in \Sigma^*$.*

Then there exists a constant c (depending on P and the choice of universal prefix machine U) such that for all $x \in \Sigma^*$,

$$K_U(x) \le -\log P(x) + c,$$

where $K_U(x)$ is the prefix Kolmogorov complexity of x.

Moreover, in general there is no constant c' such that

$$\left| K_U(x) + \log P(x) \right| \le c'$$

for all x. That is, $K_U(x)$ can be strictly smaller than $-\log P(x)$ by an unbounded amount on some strings.

Proof. Since $P(x)$ is computable and $\sum_x P(x) \le 1$, by the Kraft inequality there exists a prefix-free machine M that outputs x given an input of length $\lceil -\log P(x) \rceil$. By universality of U, there exists a constant c such that

$$K_U(x) \le -\log P(x) + c.$$

However, equality up to an additive constant cannot hold for all x. This follows from the incompressibility theorem: for any n, most strings x of length n satisfy $K_U(x) \ge n - O(1)$. Since a computable distribution P can assign probability at least 2^{-n} to only finitely many strings of length n, there exist strings for which $K_U(x) \gg -\log P(x)$. Therefore, there is no constant c' such that $|K_U(x) + \log P(x)| \le c'$ for all x. $\square$

Definition 3.4 (Algorithmic probability relative to a prefix-free machine). *Let M be any prefix-free Turing machine. The* algorithmic probability *of a finite binary string $x \in \Sigma^*$ relative to M is defined as*

$$m_M(x) := \sum_{p:M(p)=x} 2^{-|p|},$$

where the sum is over all halting programs p that output x and $|p|$ denotes the length of p in bits.

The prefix-free condition ensures that

$$\sum_{x \in \Sigma^*} m_M(x) \le 1.$$

Proposition 3.9 (Realization of a lower semicomputable semimeasure). *Let $\mu : \Sigma^* \to [0,1]$ be a lower semicomputable semimeasure, i.e. $\sum_{x \in \Sigma^*} \mu(x) \leq 1$ and $\mu(x)$ is effectively approximable from below.*

Then there exists a prefix-free Turing machine M and a constant $c > 0$ such that for all $x \in \Sigma^*$,

$$\mu(x) \leq c\,P_M(x),$$

where

$$P_M(x) := \sum_{p:M(p)=x} 2^{-|p|}$$

is the algorithmic probability of x relative to M.

Need to fix this proof c= 2? because c has to work for all.

Proof. Let $\mu : \Sigma^* \to [0,1]$ be a lower semicomputable semimeasure. By definition, there exists a computable sequence of rationals

$$\mu(x,0) \leq \mu(x,1) \leq \mu(x,2) \leq \cdots \leq \mu(x),$$

converging to $\mu(x)$ as $t \to \infty$.

Since $\sum_x \mu(x) \leq 1$, the Kraft inequality allows assigning prefix-free codewords to the increments $\mu(x,t) - \mu(x,t-1)$. For each (x,t), a binary string of length

$$\ell(x,t) := \left\lceil -\log\big(\mu(x,t) - \mu(x,t-1)\big) \right\rceil$$

serves as a codeword that outputs x.

Define a prefix-free machine M as follows:

1. Enumerate all (x,t) in the computable sequence.

2. Assign to each (x,t) the corresponding codeword of length $\ell(x,t)$ as input to M, which outputs x.

By construction,

$$\sum_{p:M(p)=x} 2^{-|p|} \geq \sum_t 2^{-\ell(x,t)} \geq c\,\mu(x),$$

for some constant $c > 0$ accounting for rounding and overhead. Hence,

$$\mu(x) \leq c\,P_M(x),$$

where

$$P_M(x) := \sum_{p:M(p)=x} 2^{-|p|}$$

is the algorithmic probability relative to M.

This completes the proof. $\qquad\square$

Definition 3.5 (Universal semimeasure). *A lower semicomputable semimeasure m is called* universal *if for every lower semicomputable semimeasure μ there exists a constant $c_\mu > 0$ such that for all $x \in \Sigma^*$,*

$$\mu(x) \leq c_\mu\,m(x).$$

Proposition 3.10 (Universal Semimeasure). *Let U be a universal prefix-free Turing machine, and define the algorithmic probability*

$$P_U(x) := \sum_{p:U(p)=x} 2^{-|p|}, \quad x \in \Sigma^*.$$

Then P_U is a universal lower computable semimeasure*: i.e. for every lower semicomputable semimeasure μ, there exists a constant $c_\mu > 0$ (depending on μ and U) such that*

$$\mu(x) \leq c_\mu\,P_U(x), \quad \text{for all } x \in \Sigma^*.$$

Proof: Let μ be any lower semicomputable semimeasure. There exists a prefix-free machine M_μ and a constant $c'_\mu > 0$ such that

$$\mu(x) \leq c'_\mu \, P_{M_\mu}(x), \quad \text{for all } x \in \Sigma^*.$$

Since U is universal, there exists a constant $d_\mu > 0$ such that

$$P_{M_\mu}(x) \leq d_\mu \, P_U(x).$$

Hence

$$\mu(x) \leq c'_\mu \, d_\mu \, P_U(x),$$

showing that P_U multiplicatively dominates μ. $\qquad\square$

Proposition 3.11 (Dominance of Universal Semimeasure). *Let M be any prefix-free Turing machine, and define*

$$P_M(x) := \sum_{p:M(p)=x} 2^{-|p|}.$$

Then there exists a constant $d > 0$ such that for all $x \in \Sigma^*$,

$$P_M(x) \leq d \cdot 2^{-K_U(x)},$$

where $K_U(x)$ is the prefix Kolmogorov complexity relative to a universal prefix-free machine U.

Proof. Treat $P_M(x)$ as a lower semicomputable semimeasure. For each finite approximation $P_M(x)|_k$, assign a prefix-free code of length $-\log P_M(x)|_k$ that outputs x (where $P_M(x)|_k$ sums the contributions $2^{-|p|}$ of all programs p among the first k that produce x within k steps, shown in Figure 9). By universality of U, this code can be simulated with fixed overhead d'_k. Taking the limit $k \to \infty$ gives

$$K_U(x) \leq -\log P_M(x) + O(1),$$

which implies

$$P_M(x) \leq d \cdot 2^{-K_U(x)}.$$

$$\square$$

Proposition 3.12 (Coding Theorem / Solomonoff-Levin Theorem). *Let U be a universal prefix-free Turing machine, and let*

$$P_U(x) := \sum_{p:U(p)=x} 2^{-|p|}$$

be its algorithmic probability. Then there exists a constant $c > 0$ such that for all $x \in \Sigma^*$,

$$\left| K_U(x) + \log_2 P_U(x) \right| \leq c,$$

or equivalently,

$$K_U(x) = -\log_2 P_U(x) + O(1).$$

The Coding Theorem establishes a deep connection between Kolmogorov complexity and algorithmic probability: strings with higher algorithmic probability have lower complexity, formalizing a precise version of Occam's razor in algorithmic terms.

Proof. Two key inequalities imply the result:

1. **Lower bound:** By definition of $K_U(x)$, the shortest program p producing x has length $K_U(x)$. Hence,

$$P_U(x) \geq 2^{-K_U(x)},$$

so

$$-\log_2 P_U(x) \leq K_U(x).$$

2. **Upper bound:** Using the universality of U, any prefix-free machine M that realizes $P_U(x)$ can be simulated on U with a constant overhead $O(1)$. This gives

$$P_U(x) \leq b \cdot 2^{-K_U(x)}$$

for some constant $b > 0$, hence

$$- \log_2 P_U(x) \geq K_U(x) - O(1).$$

Combining the two inequalities yields

$$K_U(x) = - \log_2 P_U(x) + O(1),$$

which is the Coding Theorem. □
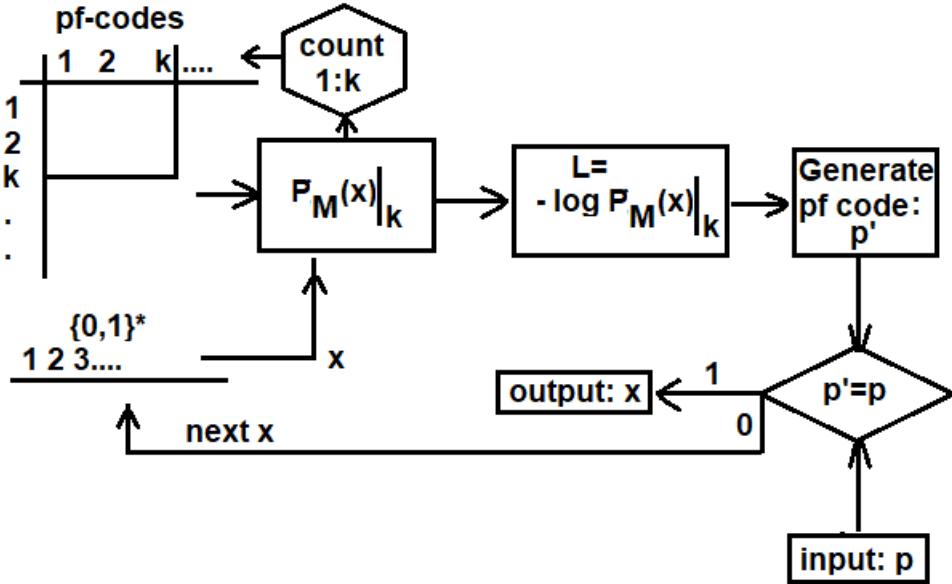
Figure 9: Construction of a Chaitin-style machine to prove $K_U(x) \leq -\log P_M(x) + C$. : *For each $x \in \{0,1\}^*$ (enumerated iteratively), execute the first k programs for k steps and identify which programs output x to compute $P_M(x)|_k$. Then take $\lfloor -\log P_M(x)|_k \rfloor$ to obtain a prefix-free code p' of that length. Compare p' with the current input p: if $p = p'$, output x; otherwise, continue iteratively with the next x.*

Often in everyday use, explanations for various phenomena begin by assigning a probability to some initial condition, followed by a derivation from that initial condition to a description of the phenomenon. Applied to algorithmic information theory, Chaitin's measure relative to a universal Turing machine U assigns an extremely small probability to a highly complex output y, since

$$P_U(y) \sim 2^{-K_U(y)},$$

and $K_U(y)$ is typically very large. This can give the impression that complex events are intrinsically unlikely.

However, there exists an alternative perspective that emphasizes a meaningful tradeoff between the probability of the initial condition and the complexity of the explanation. In this view, a complex outcome need not be improbable if it arises from structured generative processes in which simple or moderately probable initial conditions are combined through a complex explanation. Assigning a low probability to a complex phenomenon implies that any adequate explanation must be complex. In this sense, low probability is traded for explanatory complexity.

Proposition 3.13 (Probability–Explanation Tradeoff)**.** *Let U be a universal prefix-free Turing machine, and let M be a prefix-free machine. then,*

$$\max\left\{K_U(x) - |v|_{M(v)=x}\right\} \leq C_U(M),$$

where $C_U(M)$ is the description length of the machine M relative to U, signifying the **explanatory-complexity***, while $|v|_{M(v)=x}$ signifies the* **probabilistic-complexity***.*

4 Self Replication and Von Neumann Universal Constructors

A **quine** is a computer program that takes no input and produces a copy of its own source code as its only output. The simplest quine has the following characteristic structure $\langle C, M \rangle, C, M$, which, when executed, has the following properties:

- C copies $\langle C, M \rangle$: $C(\langle C, M \rangle) = \langle C, M \rangle$

- M produces C, M from $\langle C, M \rangle$: $M(\langle C, M \rangle) = C, M$

- $M, C(\langle M, C \rangle) = \langle \langle M, C \rangle, M, C \rangle$ resulting in self-replication.

This structure is not limited to computer codes, but, as will be demonstrated, it is widely utilized beyond what are commonly called computers.

4.0.1 Viruses

Definition 4.1. *(Fred Cohen) From [B3]: For every Turing Machine M and every non-empty set of Turing programs V, the pair (M, V) constitutes a **viral set** if and only if, for every $v \in V$, the following conditions hold for all computational histories of M:*

- *For all time instants $t \in \mathbb{N}$ and for every tape cell j of M:*

    1. *The tape head of M is positioned at cell j at time t,*
    2. *M is in its initial state at time t, and*
    3. *The tape cells starting at index j contain the virus v,*

    then there exists another virus $v' \in V$ such that:

    1. *At a later time $t' > t$, v' is present in the tape cells starting at a position j', where j' is sufficiently distant from the initial position j of v,*
    2. *The virus v' is written by M at some intermediate time t'' such that $t < t'' < t'$.*

Definition 4.2. *From [B3]: A virus v **evolves** into a virus v' with respect to a Turing Machine M if and only if the following conditions are met:*

$$(M, V) \in \mathcal{V} \quad and \quad v \in V \quad and \quad v' \in V \quad and \quad v \xrightarrow{M} \{v'\}.$$

Here, $(M, V) \in \mathcal{V}$ denotes that (M, V) is a viral set, v and v' are elements of V, and the notation $v \xrightarrow{M} \{v'\}$ indicates that under the execution of M, v evolves into v'.

Consider a sequence of viruses $v_1, v_2, \ldots, v_n$ such that each virus $v_i \in V$ satisfies the following evolutionary relationship:

$$v_1 \xrightarrow{M} v_2 \xrightarrow{M} \ldots \xrightarrow{M} v_n,$$

where each v_i belongs to the set V, and the transition $v_i \xrightarrow{M} v_{i+1}$ indicates that v_i evolves into v_{i+1} under the action of the Turing Machine M. A virus v' is considered an **evolution** of virus v in M if v precedes v' in such a sequence.

Given two viral sets (M, V) and (M, V'), either $V \cap V' = \emptyset$, or if not, then they share sequences such that one sequence starts at the middle of another.

Proposition 4.1. *From [B3]:Any union of a finite number of viral sets is also a viral set.*

Proposition 4.2. *(Largest Viral Set)* from [B3]: *For any Turing Machine M, if there exists a virus set V such that (M, V) is a viral set, then there exists a set U such that:*

1. *The pair (M, U) is a viral set.*

2. *For every virus set V such that (M, V) is a viral set, every virus in V is also in U.*

From [B3]: The set U is referred to as the largest viral set with respect to M, and it is denoted as LVS(M).

Definition 4.3. *Smallest Viral Set* from [B3]:

A smallest viral set SVS(M) with respect to a Turing Machine M is a set V that satisfies the following two conditions:

- *The pair (M, V) is a viral set, i.e., (M, V) ∈ 𝒱.*

- *Any subset U of V such that (M, U) is not a viral set, i.e., (M, U) ∉ 𝒱.*

In particular, the self-replicating virus $V_0 = \{v\}$ for some M such that $v \xrightarrow{\text{M}} v$ is a smallest viral set, i.e., $(M, V_0) \in \mathcal{V}$.

Proposition 4.3. *Non evolving Virus* From [B3]: *There exists a Turing Machine M such that the smallest viral set with respect to M, denoted SVS(M), is a singleton. In other words, |V| = 1 such that,* $v \xrightarrow{\text{M}} v$.

Proposition 4.4. *Undecidability of viral detection* From [B3]: *There does not exist a virus detector D that can decide whether any pair (M, V) is a viral set.*

Proposition 4.5. *There is a machine M for which every input is a virus.*

Proof. Simply the universal copier!!! $\qquad\square$

Proposition 4.6. *Undecidability of viral evolutivity* From [B3]: *There does not exist a virus evolvability detector D that can decide whether any given virus v′ has evolved from another virus v.*

Proposition 4.7. *Viral computability* from [B3]: *For every Turing program M′(x), there is a v ∈ V in a viral set (M, V) that can compute M′(x).*

The theorem states that for any Turing Machine M capable of computing a function, there exists a pair (M, V), where V is a virus or a viral-like entity, such that the computations of M can be equivalently performed by V.

This equivalence extends to sequences (binary strings) and Gödel numbers, meaning any computation done by a Turing Machine can theoretically be performed by an entity that "evolved" from a virus.

Proposition 4.8. *From [B3]: Viral Decidability is* Π_2 *complete.*

4.0.2 Mechanical Replicators

Self replication can come in simple forms. For example, a crystal can be a self replicator. They have the form aa and $a \rightarrow aa$. Penrose replicator is an interesting **kinematic replication** phenomenon. [C1] shows simple kinematic replicators of this form. [C2].

Meccano sets were among the most fascinating toys of the 20th century, allowing standardized building blocks to be assembled in various combinations to create intricate structures, such as vehicles and machines. Designing a Meccano set involves careful considerations: too few building blocks lead to

cumbersome assemblies with excessive screws, while too many reduce the flexibility to construct diverse structures from the same parts. One particularly compelling aspect is the instruction sheet, which provides step-by-step guidance for constructing each model. A central design challenge is to create a set of building blocks capable of producing a large class of valid structures—defined as assemblies in which no part is under strain, bent, or broken—and to ensure the existence of an algorithmic process for constructing each structure part by part, effectively defining completeness in this context.

Human hands are used to assemble these Meccano models, executing highly precise and intricate tasks that are often taken for granted. This raises the question of whether it is possible to construct a **Meccano constructor** machine that replicates the function of human hands, operating according to commands encoded on a tape rather than receiving instructions from the brain. If such a tape exists for constructing the machine itself, the system would possess the capability for self-replication, echoing **von Neumann's theory of self-replicating automata**, where a machine not only performs predefined tasks but can also construct an identical system by interpreting stored instructions.
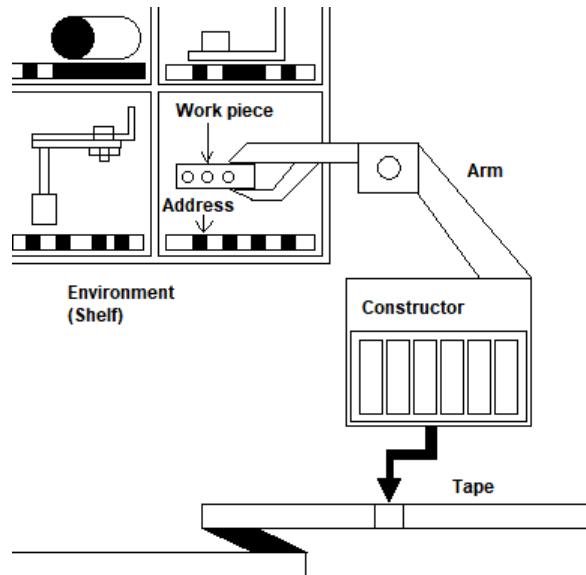
Figure 10: "Meccano" Constructor:*A plausible design would include a shelf or environment with designated addresses, allowing different parts to be systematically located based on instructions encoded on the tape.*

Human hands are marvelous machines, yet they lack the ability to construct an exact copy of themselves. As they can assemble a vast range of objects by folding into different forms, their versatility bears a striking resemblance to proteins, which also fold into various shapes to perform specific functions on molecules. Just as proteins execute biochemical operations through their structural adaptability, human hands manipulate tools and materials with remarkable dexterity, enabling complex construction and assembly processes. In this analogy, coenzymes that assist proteins in carrying out their functions resemble the tools used by human hands, enhancing their ability to perform specialized tasks. The key distinction, however, is that RNA-protein machinery possesses the capability for self-replication by following instructions encoded in DNA. Unlike human hands, which require external guidance and cannot reproduce themselves, the molecular machinery within cells operates autonomously. Ribosomes, composed of RNA and proteins, read genetic instructions to synthesize proteins, including those essential for their own function.

Life's molecular machinery can be conceptualized as analogous to two cooperating hands—one composed of RNA and the other of proteins. Just as two hands work together to manipulate objects with precision, RNA and proteins cooperate in cellular processes, each complementing the other's functions. RNA, with its ability to store and transmit genetic information, acts as both a template and a catalyst, while proteins, with their diverse structural and enzymatic roles, execute the necessary biochemical

transformations. This partnership forms the foundation of molecular biology, driving the self-replication and functionality of living systems.

4.0.3 Universal Constructors

Von Neumann's work on self-replicating machines was rooted in his fascination with automata that could replicate themselves without degenerating in complexity. His theory identified three key characteristics necessary for such machines: logical universality, construction capability, and constructional universality.

- **Logical universality**: This allows a machine to function as a general-purpose computing device, capable of simulating any Turing Machine, which is essential for reading and executing complex instructions.

- **Construction capability**: This enables the machine to manipulate materials and energy, allowing it to replicate itself.

- **Constructional universality**: Parallel to logical universality, this ensures that the machine can manufacture any machine that can be constructed from a finite set of parts, assuming an unlimited supply of these parts.

Von Neumann's design for a universal constructor was simple yet profound, consisting of four components:

1. **Constructor (A)**: A machine capable of constructing another machine from explicit blueprints.

2. **Blueprint Copier (B)**: A component that copies the blueprints for replication.

3. **Controller (C)**: The component that controls the operation of both the constructor and the copier, alternating between them to orchestrate the replication process.

4. **Blueprints** $\varphi(A + B + C)$: A set of instructions that describe how to construct the constructor, the copier, and the controller.

The replication process unfolds as follows: The controller actuates the copier to replicate the blueprint set $(\varphi(A+B+C))$, and then the constructor builds a new set of components—another $(A+B+C)$—and ties it to the newly copied blueprints. The machine then creates a second automaton identical to itself, achieving self-replication. This mechanism avoids the paradox of self-reference by ensuring that the blueprint is both copied (uninterpreted) and translated (interpreted).

Von Neumann also proposed that this system could evolve further by incorporating new features. If the machine could produce an additional object, D, besides replicating itself, it could evolve or manufacture products beyond mere self-replication. This concept parallels the cellular process of reproduction, where von Neumann's components map to biological counterparts: ribosomes (A), DNA polymerase enzymes (B), expression-control machinery (C), and DNA $(\varphi(A + B + C))$ as the blueprint. This dual use of information—interpreted and uninterpreted—was later confirmed in molecular biology, demonstrating the profound insight von Neumann had into the nature of replication and evolution. [C2].

Von Neumann's dissatisfaction with his original kinematic model led him to develop a more mathematically rigorous model using cellular automata. His initial work on this model began in the Fall of 1952 and continued into late 1953, though the full design was never completed. It was first publicly presented in the Vanuxem Lectures at Princeton in March 1953, and later published in 1966 by Burks.

The idea of a self-replicating machine described using a "cell space" format was suggested by Stanislaw Ulam in the late 1940s. This model consists of a regular grid or tessellation, with each cell occupied by a finite-state automaton that can interact with its neighboring cells in specific ways.

Von Neumann's cellular automaton model uses a checkerboard system, where each cell contains a finite-state automaton. The cells can be in one of 29 possible states, with interactions occurring only between

neighboring cells in the four cardinal directions (up, down, left, right). The behavior of each cell depends on its own state and the states of its neighbors.

At the beginning of the operation, most cells are in the "U" (unexcitable) state. Cells in the "U" state remain unchanged if all their neighbors are also in the "U" state, serving as passive elements in the system. These "U" cells can be seen as insulation or a substrate surrounding more active cells.

There are several types of active cell states:

- **Ordinary transmission states**: These cells direct their activity in the four cardinal directions, and can be in excited or quiescent modes. There are eight types of ordinary transmission states.

- **Special transmission states**: Similar to ordinary transmission states but primarily responsible for injecting signals into cells to convert "U" cells into active ones. These states can also return active cells to "U" states.

- **Confluent states**: Activated when all neighboring cells direct signals toward them. These states emit signals outward after a delay and can act as AND gates or branching elements.

Von Neumann used his cellular automaton system to design a general-purpose computing machine. He created components like:

- **Pulsers**: Emit a finite sequence of pulses upon activation.

- **Periodic pulsers**: Emit repeated pulses until signaled to stop.

- **Decoders**: Detect specific patterns of pulses.

Using these components, he designed the control portion of a computing machine and an adjacent memory unit. The memory unit stores information and can be accessed by the control unit.

For the construction process, Von Neumann designed a construction unit, which, based on instructions from the memory unit, sends out a constructing arm that creates a pathway of transmission cells into a region of "U" cells. The arm then converts the "U" cells into the cell types specified in the memory.

The construction unit is capable of creating any passive configuration of cells, making it a universal constructor. Since the parent machine itself is created in a passive form, it can replicate itself. The replication process follows these steps:

1. The parent machine receives instructions to construct a duplicate of its control, construction, and memory units.

2. The parent machine then copies the instructions into the memory of the newly constructed machine.

3. The offspring machine is activated and becomes a duplicate of the parent machine, completing the self-replication process.

This cellular model of machine replication meets the logical properties required for self-replication, including **logical universality**, **construction capability**, and **constructional universality**, demonstrating the potential for cellular automata to replicate themselves.

A computation refers to the activity performed by a computer when provided with an input. A mineral catalyst can be regarded as a computer, with the chemical reactions it facilitates serving as computations. Computations are not limited to traditional tapes; for instance, they can be carried out on objects such as a torus, which can be computationally deformed into a cup. Similarly, molecules of reactive gases within a reactor chamber can be interpreted as symbols, with the three-dimensional volume acting as an analogue to the computational tape. In such cases, the fields and physical laws governing the behavior of the system function as the computer.

This perspective becomes particularly significant when considering that Life itself fundamentally operates as a computational system. Life employs genetic programs, initiated by the action of RNA polymerase on DNA. These programs correspond to proteins, and the intricate processes of transcription, translation, and replication collectively render Life a naturally formed universal constructor. Therefore, computers are not merely human-made machines but can arise naturally as a consequence of physical laws. This realization underscores the need to generalize computational theory to encompass such naturally occurring phenomena.

Cell Component	Universal Constructor Component
RNA Polymerase	Tape Reader
Ribosome	Constructor Arm
DNA Replicase	Universal Copier
Proteins	Machines

Table 1: Comparison between Cell Components and Universal Constructor Components
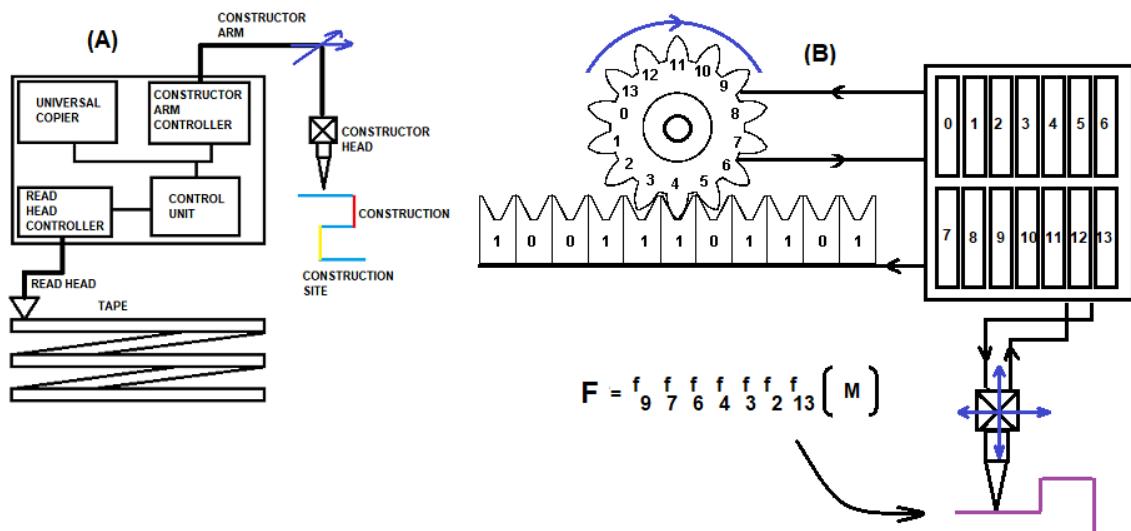
Figure 11: Von Neumann's Universal Constructor: *(A) A **tape** containing the machine's description and construction instructions, a **reading head** for interpreting the tape, a **constructor arm** for assembling components, a **universal copier** for replicating the tape, and a **control unit** for coordinating the entire process. The control unit translated the instructions from the tape into actions for the constructor arm, ensuring precise placement of each building block.(B) A schematic implementation for the Von Neumann's Universal Constructor, based on [C3]. The wheel functions as a program counter, advancing along the tape in a manner analogous to a rack-and-pinion system. In this setup, the tape serves as a sequence of instructions, marked with 1s and 0s, which dictate whether a given program should be executed or skipped. The program counter, represented by the pinion, has numbers inscribed on its teeth, corresponding to specific program steps. As the pinion engages with the tape, it interprets the encoded instructions, guiding the system's operation in a structured and deterministic manner. The programs themselves are hardwired into the system and stored within a program bank, which the program counter accesses as needed. These pre-defined programs govern essential operations, such as the movement of the constructor arm and the motion of the tape reader, ensuring coordinated and precise execution of automated tasks. This architecture enables complex, rule-based control over the system's functionality while maintaining a clear, mechanical analogy to computational execution.*

5 Generalized Computational Theory

A **generalized computational theory** is essential for describing systems that exhibit computational characteristics but do not conform to traditional computer models. Such a theory allows for the exploration of processes and behaviors in systems that process information, adapt, or evolve in ways similar to computation, without necessarily being digital or following the rigid structure of conventional computing.

The essence of generalized computational theory is that systems exhibiting computational behavior or resemblance to computers may not present themselves as traditional computing devices. Life, in its most intriguing case, serves as an example. However, despite their unconventional nature, such systems can be formally described in detail. These descriptions take the form of strings, which, in the traditional sense, are computational entities. Consequently, conventional computational theory applies to them. For instance, while a living biological cell cannot be classified as a computer or a computational system in itself, a detailed description of the cell being a string, constitutes a computational object. In this way, computational theory extends beyond abstract machines and can be applied to real-world entities, such as living cells.

Traditional computational theory, as outlined in the preceding sections, primarily focuses on the manipulation of strings and tape-based models. However, not all computational systems operate within this framework. The *Meccano constructor* previously discussed serves as an illustrative example of such a system. Life, for instance, resembles a computational phenomenon akin to a natural computer, yet it does not process strings or tapes but rather molecules. Its computations are embodied in chemical reactions. This divergence underscores the need for a more general computational theory capable of addressing such systems and their unique operational paradigms.

Given the well-established theoretical framework for strings, one approach to address this situation is to focus on the descriptions of arbitrary systems rather than the systems themselves, as these descriptions can be represented as strings.

Thus, the generalized computational theory can be viewed as a theory of computations over descriptions. This approach draws inspiration from how humans structure their understanding of the world. Humans assign arbitrary words or strings to objects and phenomena, then describe these phenomena in terms of what they consider fundamental objects or processes. These descriptions, formulated in a computational framework, serve as explanations of the observed phenomena.

*If the following definition might be intimidating. Refer to **Figure 9**.*

Definition 5.1. *A generalized computational system* *is a tuple* $G \equiv \langle \mathcal{J}, \mathcal{A}, \Gamma, I_\Gamma, I_\Gamma^{-1} \rangle$, *where* $\mathcal{J}$ *denotes a set of **phenomena**, understood as abstract entities that may admit string descriptions.*

The component $I_\Gamma = (Q \oplus \Omega, \Lambda)$ *is called an **interpreter**. It is a hypothetical machine consisting of a computational component* Q, *possessing computational power equivalent to a Turing machine, an oracle function bank* $\Omega = \{\omega_1, \ldots, \omega_n\}$, *and a special operation called **reconstruction** Λ, which utilizes* $Q \oplus \Omega$.

Given an input string $s \in \mathcal{A}^*$, *the interpreter* I_Γ *may, through the reconstruction operation* Λ, *produce a phenomenon* $J \in \mathcal{J}$ *if such a reconstruction is possible. Otherwise, the reconstruction produces no result. The set of strings for which reconstruction is defined constitutes the **language** $\Gamma \subseteq \mathcal{A}^*$.*

*The **reverse interpreter** I_Γ^* is a canonical mapping that assigns to each phenomenon* $J \in \mathcal{J}$ *a unique description* $\gamma \in \Gamma$. *It operates over a standard, possibly infinite, set of keywords together with internal generative rules, ensuring that each phenomenon admits exactly one canonical description. Its inclusion is necessary for the operation of* G, *as it enables unambiguous comparison of phenomena at the level of description.*

The fundamental purpose of a generalized computational system is to determine whether two descriptions $\gamma, \gamma' \in \Gamma$ *are **isomorphic**, conditioned by :*

$$\gamma \cong \gamma' \iff I_\Gamma^* I_\Gamma(\gamma) = I_\Gamma^* I_\Gamma(\gamma').$$

Intuitively, the reverse interpreter I_Γ^* is analogous to the combination of human consciousness with technological tools that allow humans to create descriptions of complex phenomena. On the other hand, I_Γ is a hypothetical machine capable of recreating these phenomena.

Definition 5.2. *Isomorphic Phenomenon:* $J \sim_G J'$ when $I_\Gamma^*(J) = I_\Gamma^*(J')$.

Definition 5.3. *Isomorphic Descriptions:* If $\gamma, \gamma' \in \Gamma$ and $I_\Gamma(\gamma) \sim_G I_\Gamma(\gamma')$, then γ is **isomorphic** to γ' and, denoted by $\gamma \sim_G \gamma'$.

Definition 5.4. *Isomorphism Class:* The set $\mathcal{C}_J = \{\gamma \mid \gamma \sim_G I_\Gamma^*(J)\}$ is an isomorphism class of all descriptions that describe J.
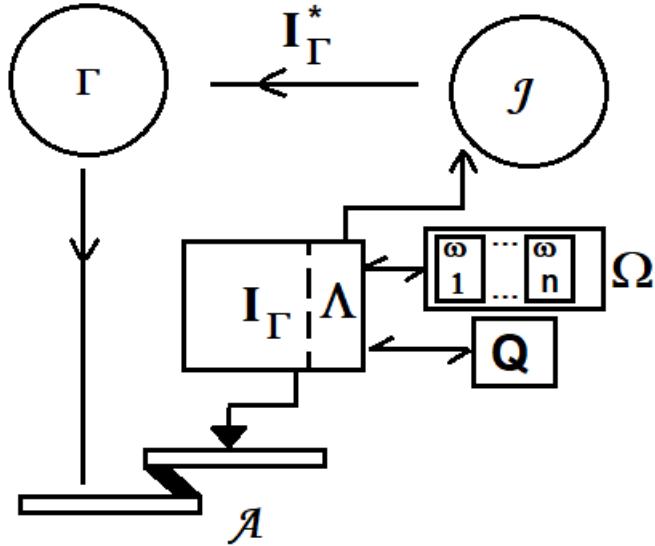

Figure 12: The Generalized Computational System: *The reverse interpreter I_Γ^* can map certain phenomena in $\mathcal{J}$ to strings of alphabet $\mathcal{A}$. While phenomena are abstract and inherently non-computational—since they cannot be generated by a Turing Machine—strings are computational, being the sole entities that a Turing Machine can produce. Thus, the reverse interpreter transforms non-computational phenomena into computational objects, specifically strings. Utilizing the oracles Ω and Q, with equivalent computational power to that of a universal Turing Machine, the interpreter I_Γ is capable of producing phenomena in $\mathcal{J}$ from their descriptions in Γ. Essentially, traditional computer code, such as that written in C++, can be interpreted as descriptions of the programs it defines. The generalized computer extends this concept by introducing a more comprehensive framework that accommodates entities not traditionally classified as computer programs but that exhibit analogous behavior.*

It is often necessary to compare one phenomenon with another. Logically, comparing two phenomena is simply an act of comparing their descriptions. If the two descriptions are similar, then it is intuitive to conclude that they refer to the same phenomenon. Cars of the same make and model will have the same description, while cars of different makes and models will share some similarities. However, the degree of similarity is determined computationally when their descriptions are compared. When there is a common description for multiple phenomena, these phenomena can be grouped into a class of their own, characterized by the shared description.

Descriptions are represented as strings, which can encode functions over other strings. More precisely, a string arranges symbols from an alphabet into a sequence, forming an atomic string, which can then serve as a building block for constructing more complex strings.

This suggests that all descriptions $\gamma \in \Gamma$ are of the form

$$\gamma = f(\{x_i\}, \{e_j\})$$

where $x_i \in \Gamma$ are other strings, and $e_j \in \mathcal{E}$ are extra parameters from a set $\mathcal{E}$ used by some function $f : \mathcal{A}^* \times \mathcal{P}(\mathcal{E}) \to \Lambda$ to construct γ.

Ultimately, it can be asserted that all $\gamma \in \Gamma$ take the form

$$\gamma = f(\{\gamma_{\omega_i}\}, \{p_j\}),$$

where γ_{ω_i} represent descriptions for $\omega_i \in \Omega$, with some function $f : \mathcal{P}(\Gamma_\Omega) \times \mathcal{P}(\mathcal{E}) \to \Lambda$, and Γ_Ω represents all possible γ_{ω_i} for $\omega_i \in \Omega$.

Similar to how programmers develop code using high-level languages, for a given $\gamma \in \Gamma$, the expression for γ can be structured with modularized sub-functions $h_1, \dots, h_a$ such that

$$\gamma = f'(h_1, \dots, h_a, p'_1, \dots, p'_b) = f(\gamma_{\omega_1}, \dots, \gamma_{\omega_n}, p_1, \dots, p_m).$$

This formulation encapsulates the hierarchical composition of functions, where higher-level abstractions $\{h_j\}$ are constructed from underlying oracle components γ_{ω_i} and parameters, ensuring a structured approach.

Definition 5.5 (Generator Abstraction)**.** *Let $\mathcal{J}$ be a set of phenomena. If there exists a set $\{h_i\}$ and a function f such that for every $J \in \mathcal{J}$ there exists a parameter set $\{e_j\} \subset \mathcal{E}$ satisfying $\lambda_J \sim f(\{h_i\}, \{e_j\})$, then $\{h_i\}$ is called a **set of generators** for $\mathcal{J}$, the elements h_i are called **generators**, and f the **generating function**.*

Definition 5.6. *The **generalized Kolmogorov complexity** $K_G(J)$ of a phenomenon $J \in \mathcal{J}$ is given by $\min |\gamma|$ such that $\gamma \in \mathcal{C}_J$.*

Generalized Kolmogorov complexity enables the theoretical definition of complexity for arbitrary phenomenon.

Definition 5.7. *A phenomenon $J \in \mathcal{J} \in G$ with a description $\gamma_J \in \Gamma$ is said to be G-computational if $I_\Gamma \in G$ produce $I_\Gamma(\gamma_J) \sim_G J$.*

When J is said to be computational with respect to a hypothetical I_Γ, it is computational in the general sense. If no generalized computational system G for describing J exists, for which $I_\Gamma \in G$ is equivalent to a Turing Machine, then J is not computational in the Turing sense. In this generalized framework, the meaning of being computational must be handled with respect to the context of G being considered.

A phenomenon may not be computational in the Turing sense, meaning that it cannot be reconstructed from a string description input into a Turing Machine. In fact, only those phenomena that inherently involve strings can be computationally created in Turing sense. Other phenomena require hypothetical interpreters equipped with a characteristic oracle function bank for their reconstruction. However, even phenomena that are solely based on strings may still be non-computational in the Turing sense, as there exist functions on strings that are inherently non-computable within the Turing framework.

Definition 5.8. *A **time-ordered phenomenon** in some G is a $J \in \mathcal{J}$ such that*

$$J \equiv (J[0], \dots, J[T], \prec),$$

where each $J[t] \in \mathcal{J}$ and $\prec$ is an ordering relation such that $J[t] \prec J[t']$ whenever $t \prec t'$.

An informative description can be understood as a string that aids in answering a predefined list of questions when provided to a universal Turing Machine. Intuitively, this suggests that, given such a description, a Turing Machine can manipulate the string to derive answers to the specified questions.

Given a *list of questions* or query X on the phenomenon J, a corresponding Y can be obtained as the answers to the query X on J by analyzing the function f that corresponds to the description γ_J. In the case that J is a time-ordered phenomenon, a time ordered sequence $Y[t]$ can be obtained by analyzing X on $\gamma_{J[t]}$.

Without access to the interpreter I_Γ, Y may not be accurately determined. Let $P(Y \mid X, \gamma_J)$ denote the probability distribution of the random variable Y when X is analyzed on γ_J using only a universal Turing Machine, and let $P(Y \mid X, \gamma_J, I_\Gamma)$ denote the probability distribution of Y when X is analyzed on γ_J with access to the interpreter I_Γ.

The closer $P(Y \mid X, \gamma_J)$ is to $P(Y \mid X, \gamma_J, I_\Gamma)$, the more Turing wise computational the phenomenon J is, given γ_J. Let there be a measure of their **informational difference** denoted by $D(Y \mid X, \gamma_J, G)$.

*In the above definition, the base computational device used is the universal Turing Machine. However, in certain situations, it may be more appropriate to use a **universal quantum Turing Machine** instead. Nevertheless, this substitution does not alter the fundamental logic.*

It is particularly interesting that probability distributions are indeed a natural place for uncomputable functions to hide themselves. For instance, those probability distributions whose distribution function is uncomputable.

Definition 5.9. *The **ideal generalized computational system** G for a given X and J is the one that minimizes $D(Y \mid X, \gamma_J, G)$, by finding a corresponding γ_J.*

Definition 5.10. *An X for which $D(Y \mid X, \gamma_J, G) = 0$ for J is said to be **Turing-adequate** for J. The **Turing-adequate class** $\mathcal{C}_X$ for G is defined by*

$$\mathcal{C}_X = \{J \mid \exists_{\gamma_J} \text{ such that } D(Y \mid X, \gamma_J, G) = 0\}.$$

Definition 5.11. *Analyzing X for a **time-ordered** phenomenon $J \equiv (J[0], \ldots, J[T], \prec)$ would yield a similarly time-ordered sequence of $Y[t]$ corresponding to $J[t]$. If there exists a Turing program E such that*

$$Y[t] = E(\gamma_J[0], t),$$

and

$$D(Y[t] \mid X, \gamma_J[t], G) = 0$$

*for every t, then X is said to be **Turing computational** for J.*

*The class $\mathcal{C}_{X,E} = \{J \mid \exists_{\gamma_J} \text{ such that } \forall_t D(Y[t] \mid X, \gamma_J[t], G) = 0 \text{ and } Y[t] = E(\gamma_J[0], t)\}$ is called a **Turing computational class** in G.*

The Turing computational class $\mathcal{C}_{X,E}$ is a profoundly intriguing construct. It comprises a set of phenomena $\{J\}$ that embed a Turing computational system. Notably, in case E is a universal Turing Machine and all Turing programs within E are embedded in $\mathcal{C}_{X,E}$, then $\mathcal{C}_{X,E}$ effectively constitutes a system capable of implementing E. Moreover, this implies that replicators and evolving viruses that are characteristic of Turing Machines, are necessarily embedded within it, underscoring its fundamental significance.

In the real world, there are no physical Turing Machines; rather, there exist material devices composed of plastic, silicon, and metals that embed a Turing Machine within them. Similarly, life exhibits characteristics suggesting that it possesses sufficient Turing-like computational power and resembles the von Neumann universal constructor, as outlined in **Table 1**. The material world consists of non-computational phenomena; however, these non-computational phenomena, such as fundamental particles, can have Turing-like computational power embedded within them. The formalism in $\mathcal{C}_{X,E}$ generalizes this idea in a mathematical sense.

The implementation of E within a class $\mathcal{C}_{X,E}$ intuitively implies the presence of a Turing Machine-equivalent system embedded within $\mathcal{C}_{X,E}$. This is particularly intriguing because highly complex or interesting phenomena, such as life, can emerge within such a system if E possesses sufficient computational power to implement them. Essentially, all behaviors observed in life, including the intricate operations of cellular machinery, can be understood as computational processes. If life itself functions as a computational system with embedded computational power, its capacity to manifest such complexity is unsurprising.

Universal Turing Machines and **universal constructors** appear to share a fundamental conceptual similarity. A Universal Turing Machine M is a Turing Machine that takes an input string encoding

another Turing Machine M' and simulates its execution. Conversely, a Universal Constructor can be regarded as a machine M that takes an input description x and constructs another machine M'. Essentially, *simulate* and *construct* emerge as distinct attributes or specialized functions of M, respectively. These ideas can be further formalized.

Definition 5.12. *Universality and Completeness: Let the (f, M) relation class $\mathcal{C}_{(f,M)}$, where f is a function and $M \in \mathcal{J}$ is a phenomenon, be defined as*

$$\mathcal{C}_{(f,M)} = \{M' \mid \exists_x \text{ such that } \gamma_{M'} \sim_G f(\gamma_M, x)\}.$$

*If $M \in \mathcal{C}_{(f,M)}$, then M is said to be **universal** for the class $\mathcal{C}_{(f,M)}$, and $\mathcal{C}_{(f,M)}$ is said to be **complete** over f.*

If $\mathcal{K} \subset \mathcal{J}$ is a set of phenomenon, then define

$$\Gamma_{\mathcal{K}} = \{\gamma \mid \gamma \sim_G I_\Gamma^{-1}(J) \text{ such that } J \in \mathcal{K}\}$$

as the set of descriptions for elements J in $\mathcal{K}$.

Given two relation classes $\mathcal{C}_{(f,M)}$ and $\mathcal{C}_{(f',M')}$, if there exists a one-to-one mapping

$$T : \Gamma_{(f,M)} \to \Gamma_{(f',M')}$$

such that T is Turing-wise computational, then the two relational classes are Turing-wise computationally equivalent with respect to G.

For example, a Universal Constructor and a Universal Turing Machine can be computationally equivalent. Despite the fact that the Universal Constructor has a function construct given by f and the Universal Turing Machine has a function simulate given by f', if there exists a computational mapping between them, then they are fundamentally equivalent, representing the same underlying concept under different names. Universal constructors in cellular automaton systems can construct Turing Machines, and Turing Machines can simulate cellular automata. This enables the existence of such a mapping T between universal computers and universal constructors.

The essence of generalized computational theory is that systems exhibiting computational behavior or resembling computers may not necessarily present themselves as traditional computing devices. Life, in its most intriguing case, serves as an example. However, despite their unconventional nature, such systems can be formally described in detail. These descriptions take the form of strings, which, in the traditional sense, are computational entities. Consequently, conventional computational theory applies to them. For instance, while a living biological cell cannot be classified as a computer or a computational system in itself, but a detailed description of the cell—being a string—constitutes a computational object. In this way, computational theory extends beyond abstract machines and can be applied to real-world entities, from commercial computers to living cells. This has been established through Definition 24 and Definition 25.

If a highly sophisticated phenomenon exists and obeys a computational explanation, it follows that the system inherently possesses an embedded computational structure, as formalized within generalized computational theory. Conversely, a system that contains a computational structure of sufficient power inherently has the capacity to leverage this computational capability to generate sophisticated and complex behavior that obeys a computational explanation.

It is important to recognize that Universal Turing Machines are theoretical constructs, whereas any physically realizable Turing Machine must necessarily operate with a finite tape, and therefore cannot be truly universal. The distinction arises from physical limitations: in practice, no machine can store or process an unbounded amount of information. However, the concept of universality can still be meaningfully approached by considering an asymptotic perspective. If the tape length were allowed to grow without bound, then a physical Turing Machine could, in principle, emulate the behavior of a Universal Turing Machine. In this sense, universality can be interpreted as an asymptotic property, where finite machines approximate universality more closely as their available resources increase.

Definition 5.13. *Asymptotic Isomorphism: Given G, let $M, M' \in \mathcal{J}$ be phenomena. If there exists f such that*

$$\gamma_{M'} \sim_G \lim_{k \to \infty} f(\gamma_M, k),$$

then M is said to be asymptotically isomorphic to M' in G.

5.1 Applications to the Phenomenon of Life

Explaining life and its origin is a prominent topic in contemporary discourse. However, it is perhaps more intriguing to consider what such an explanation truly entails. An explanation of the origin and function of life inherently requires a computational framework capable of simulating both the emergence and functions of life based on this explanation. This computational model must encompass the processes that allow life to originate and sustain itself, effectively capturing the essence of life's mechanisms in a formal, algorithmic form. In simple terms, this explanation, utilizing modular structures and abstract functions to represent different biological processes, is analogous to computer program code that, when input into a hypothetical machine, can simulate the origin and functioning of life.

Formally, within the context of generalized computational theory, life is comprised of a set of phenomena $\mathcal{J}$, which can be described by a language Γ through a hypothetical interpreter I_Γ. This interpreter is capable of reconstructing $\mathcal{J}$ from Γ. The computational properties of life can be analyzed by studying the elements $\gamma_J \in \Gamma$, focusing on their modular structures that represent distinct computational processes within biological systems.
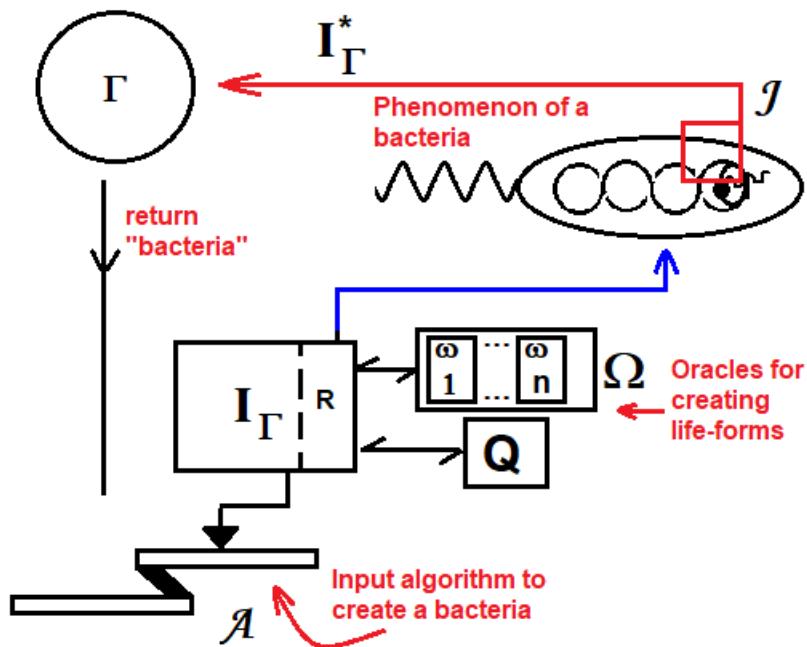
Figure 13: Life as a computational phenomenon: *The reverse interpreter I_Γ^* generates a description $I_\Gamma^*(J)$ of life-form J, which, when input into the Interpreter I_Γ, is able to reconstruct it by utilizing its oracle function bank Ω. The computational properties of the description class $C_J = \{\gamma | \gamma \sim_G I_\Gamma^*(J)\}$ represent the computational nature of life as descriptions are represented as strings, which can be analyzed as computational entities.* **Life forms are universal constructors in the generalized computer that create life.**

Commercial computers, in their actual physical realization, are composed of atoms, subatomic particles, and associated fields arranged in such a way that they are capable of producing computations. This

implies that a computer necessarily contains components that are not themselves computational in nature—namely, the physical substrate whose exact instantiation cannot be fully captured by computation alone and would require oracles or idealized assumptions to be reproduced exactly.

At the same time, certain aspects of a computer's behavior can be abstracted and explained entirely in terms of symbolic string manipulation, as formalized by the Turing machine model. It is this computational abstraction that is of primary interest to mathematicians, whereas physicists may seek to understand how the non-computational physical processes give rise to, constrain, or interact with the computational ones.

Essentially, an arrangement of physical matter may be called a computer because there exists a concise description that captures its relevant computer-like behavior. i.e., by performing computational operations on this description, all properties that are relevant to the theoretical computer can, in principle, be determined. There would exist a Turing machine that, using this description, can simulate its behavior.

The same may be said of a desk, a chair, a cat, a bacterium, or even life itself. In such cases, however, there is no canonical theoretical description, and the choice of model is necessarily observer-dependent. Furthermore, since natural systems may exhibit probabilistic behavior, the adopted computational description may naturally take the form of a stochastic or probabilistic model.

The computational description of a bacterium, for instance, to a large extent resembles von Neumann's universal constructor. In this analogy, DNA plays the role of the instruction tape, RNA polymerase acts as the tape reader, the ribosome functions as the construction apparatus, and proteins correspond to the machines being produced. At the same time, such a description will also bear strong similarities to the computational abstraction of a commercial computer, since the mechanisms by which genetic programs are activated and deactivated closely resemble the control and scheduling of processes in modern processors.

Generator abstraction is particularly visible in the phenomena of life. At the most basic level, all life that can be described is describable in terms of the chemistry of molecules, making molecules the lowest-level abstraction. Amino acids combine to form proteins, nucleotides form the nucleic acids RNA and DNA, and cells assemble into multicellular organisms. Each level can be viewed as a generator of higher-level structures or behaviors, providing a natural hierarchy of abstractions.

In this analogy, atoms serve as the generators of molecules, amino acids serve as the generators of proteins, nucleotides serve as the generators of nucleic acids, and cells serve as the generators of multicellular organisms.

5.1.1 Remark on Consciousness

Phenomenon of consciousness is elusive from a purely computational viewpoint. Few researchers consider consciousness to be a passive byproduct, but most agree that it must serve a functional role. There must be an evolutionary advantage to having consciousness. The most prominent contemporary schools of thought on consciousness are probably **Integrated Information Theory** (IIT) and the **Global Workspace Theory** (GWT). IIT suggests that consciousness arises from the integration of information, without which the firing of individual neurons carries no meaningful content. GWT proposes that this integrated information is broadcast back to different regions of the brain that process it, enabling flexible, coordinated behavior.

From a computational perspective, consciousness is essential because it imparts meaning to symbols in code and enables the translation of internal conscious thoughts into symbolic representations that can be processed computationally. In other words, consciousness allows abstract symbols, such as 1s and 0s, to acquire meaning, while simultaneously providing a mechanism to encode subjective mental states into a form usable by computational systems. This dual role of consciousness — assigning meaning and encoding thoughts — lies at the core of the computational process.

In a manner of speaking, the generalized computer performs a task analogous to consciousness. It assigns meaning to an input string and then outputs a representation of what it has construed, in the form of an

output string. **Integration** in consciousness studies refers to how separate mental processes are bound together to form a unified field of experience. In generalized Computational Theory, integration can be found through isomorphism relations of the form

$$\lambda_J \sim f(\lambda_{J_1}, \ldots, \lambda_{J_n}, \ldots),$$

. This relation captures integration as the formation of a unified whole from interacting parts.

*A limited but a useful definition of **consciousness** is the capacity to assign meaning to a string or sequence of symbols.* Under this definition, a generalized computer would possess this form of consciousness.
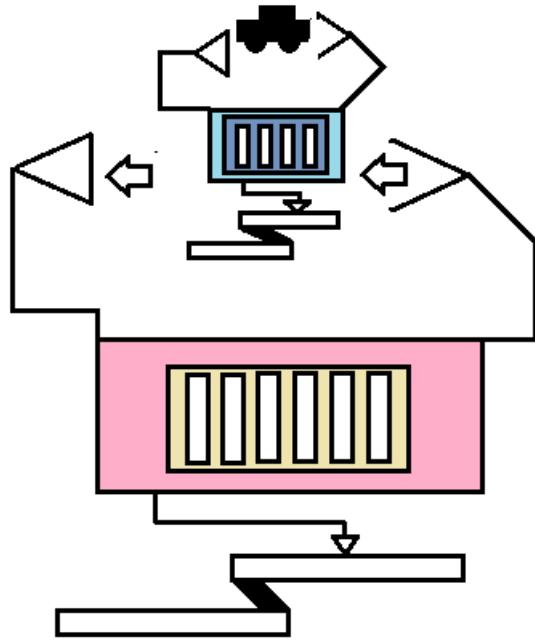
Figure 14: Universality in generalized computational theory: *A sufficiently powerful generalized computer G can recreate number of other generalized computers, and can also recreate itself when provided with its appropriate description. Collectively they form a **complete class** over the action of G.*

In generalized computational theory, **universality** is defined relative to classes of generalized computers. A **universal generalized computer** is one that can interpret descriptions of all generalized computers within the particular class to which it belongs.

When the notion of **universality** is applied to generalized computational theory and to the phenomenon of a generalized computer itself, a fascinating connection to consciousness emerges.

Metaphorically speaking, if the universe is analogous to a universal generalized computer, then the other generalized computers it can interpret are analogous to conscious beings, such as humans.

The identification of a **self** is not a necessary condition for consciousness. There exist mental disorders in which the sense of self is diminished or absent, yet the process of information integration that supports conscious experience remains intact.

Self-consciousness *is consciousness accompanied by a notion of the **self**.*

Just as a universal constructor is the computational model associated with life, the `SELF` variable can be seen as the computational model associated with **self** within **self-consciousness**. In this view, a self-aware conscious experience corresponds to the occurrence of the `SELF` within a generalized computational system.
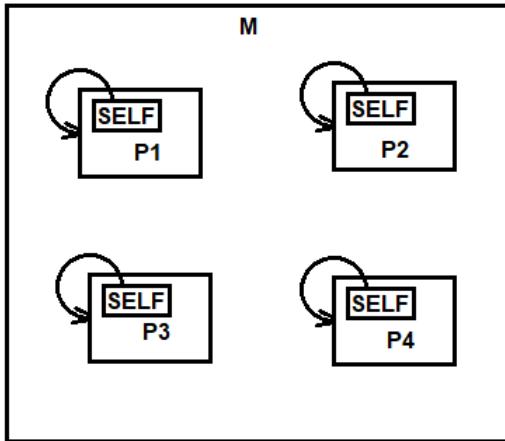
Figure 15: SELF in generalized computational theory: *It is entirely consistent for a master program(M) to lack any notion of self—that is, to have no access to a usable description of its own description—while simultaneously containing subprograms (P_1, P_2, P_3, P_4) that do possess such access, through an internal representation or variable such as SELF.*

A meaningful sense of self can be formally characterized by a function $f(\text{SELF})$. The following theorem holds for any generalized computer.

Proposition 5.1 (Undecidability of SELF-Detection Theorem). *There is no universal decider D that, given an arbitrary program P, can decide whether P contains a function of the form $f(\text{SELF})$, i.e., a function that operates on a description of P itself.*

Proof. Assume, for the sake of contradiction, that such a universal decider D exists, then construct an undecidable program M defined as follows:

$$M = \langle \text{if } D(\text{SELF}) \text{ then remove } f(\text{SELF}), \text{ else add } f(\text{SELF}) \rangle$$

□

It asserts that: *consciousness is fundamental, while self-consciousness is emergent.*

5.1.2 Remark on Observers

Observers give meaning to descriptions. Observers can be modeled as generalized computers, where internal representations of both external and internal states are constructed as phenomena. These observers may or may not possess a notion of self.

It follows that the universe may contain observers other than self-conscious entities such as humans. The universe must possess intrinsic meaning independent of human self-consciousness. The universe itself could be said to have access to its own description SELF—that is, to observe itself—although the degree and structure of any resulting self-consciousness are unknown. However, the universe need not possess a level of self-consciousness analogous to that of humans.

6   The Jigsaw Puzzle Model

Von Neumann Universal Constructor used a input instruction tape which dictated the right sequence by which primary builtin operations had to be carried out in order to construct output machines, and functioned as a deterministic machine. The idea is to create a complex machine by the possibility of its construction being decomposed in to a sequence of simple instructions.

Von Neumann's Universal Constructor employs an input instruction tape that specifies the precise sequence by which a set of primitive operations must be executed in order to construct an output machine. The constructor functions as a deterministic system, relying on the idea that a sufficiently complex machine can be built by decomposing its construction into a sequence of simple, well-defined instructions.

Cellular automata models, such as von Neumann's Universal Constructor, are often presented as demonstrations that simple local rules can give rise to complex global behavior. The intended implication is that life itself might arise from similarly simple underlying dynamics. While these models are mathematically remarkable, they rely on an initial configuration that is already highly structured. In contrast, the origin of life is generally believed to have occurred under far more chaotic and weakly organized initial conditions.

From the perspective of Kolmogorov complexity, this distinction is critical. Algorithmic complexity cannot be eliminated; it can only be relocated. If a structure exhibits high complexity, then by definition it cannot be generated from a significantly simpler description without that complexity being encoded elsewhere. In cellular automata, although the update rules may be simple, the complexity is embedded in the initial configuration.

If life admits a sufficiently simple description, then by definition it is not complex. Conversely, if life is complex, then any faithful description that captures its essential structure must itself be complex, and the degree of this complexity provides a measure of how complex life is. This follows directly from the notion of **algorithmic (Kolmogorov) complexity**.

If the elementary steps underlying life are themselves simple—for example, if they are governed by relatively simple chemical rules—then the source of biological complexity cannot reside in these rules alone. Instead, the complexity must be encoded in the manner in which molecular components are organized and interconnected.

A useful toy model for understanding how simple assembly rules can give rise to a complex object is the **jigsaw puzzle**. The rules governing assembly are trivial: pieces may be joined only when their shapes and local features are compatible. These rules are simple, uniform, and easily describable. Nevertheless, the completed puzzle typically exhibits a high degree of global structure.

Crucially, the complexity of the final image does not originate from the assembly rules themselves, but from the information encoded in the individual pieces and in their specific compatibilities. The assembling rules merely constrain allowable connections; they do not determine the global configuration. In algorithmic terms, the Kolmogorov complexity of the completed puzzle is dominated by the information content of the pieces and their arrangement, rather than by the simplicity of the assembly procedure.

This illustrates a general principle: simple local rules can facilitate the construction of complex structures, but they do not generate complexity unless that complexity is already present in the components or their organization. The jigsaw puzzle thus serves as a concrete analogy for biological assembly processes governed by simple chemical interactions yet exhibiting highly complex global behavior.

A Jigsaw Puzzle works by being able to match details at the boundary of different pieces, such as the notch patterns, the colors, and the curvature, and often without any idea of what the internal details might be. This inspires to think of a different kind of a constructor that uses a computational procedure to assemble a complex machine by combining individual parts in the same way a Jigsaw Puzzle does.
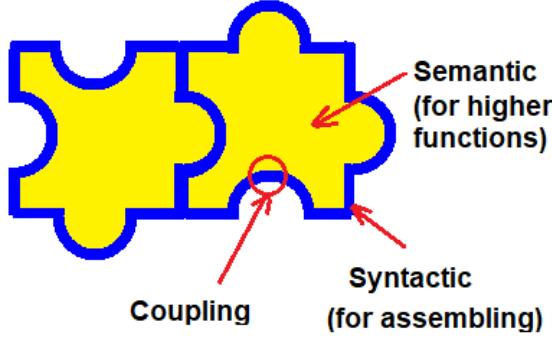
Figure 16: Jigsaw Puzzle: *The jigsaw puzzle exhibits complexity on three levels. First, the complexity involved in matching and assembling the border pieces, which can be called the **syntactic complexity**. Second, the complexity of the internal content, which can be referred to as the **semantic complexity**. Third, the **coupling complexity** in the way syntactic information and semantic information are coordinated and matched to one another. Any computational machine tasked with completing the puzzle would need to implement programs to handle each of these aspects.*

Now consider a different approach to a constructor that is subjected to non-determinism which construct its output by assembling what ever is possible from a given random input which consist of randomly ordered pieces. The input consist of a mixture of parts belonging to all possible independent output structures. Instead of following commands from well defined input instructions, the random pieces themselves would determine to what they should connect to, quite analogous to a Jigsaw Puzzle. The output is a distribution of independent structures. It is as if constructing a multiple independent Jigsaw Puzzles when their pieces are given in a mixture.

The underlying idea is relatively straightforward. Consider a **Sequentially Constructible Assembling Set** A, consisting of an enumerated list of sets of segments of computable cardinality, together with an algorithm Π that specifies a procedure for sequentially assembling a sequence by iteratively adding segments in a prescribed order. This is analogous to an enumerated list of jigsaw puzzles, each with its own collection of pieces, all governed by a single assembly algorithm. For simplicity, construction is one-dimensional, without any loss of generality.

Definition 6.1 (Sequentially Constructible Assembling Set). *An assembling set $A = (\mathcal{V}, \Pi, \mathcal{X})$ consists of:*

1. *A set of segments $\mathcal{V}$,*

2. *A sequential assembly operation Π,*

3. *A set of assembled ordered sequences $\mathcal{X}$,*

such that the following conditions hold:

1. ***Assembly:*** *For every $X \in \mathcal{X}$, there exists a finite set of segments*

$$\{v_1, v_2, \ldots, v_{W(X)}\} \subset \mathcal{V}$$

satisfying

$$X = (v_1, v_2, \ldots, v_{W(X)}),$$

where $W(X)$ is the number of segments in X.

2. ***Disjointness:*** *For any two distinct sequences $X, X' \in \mathcal{X}$,*

$$X \neq X' \implies \{v_i : v_i \text{ appears in } X\} \cap \{v'_j : v'_j \text{ appears in } X'\} = \varnothing.$$

3. **Sequential Constructibility:** *The operation Π can construct sequences one segment at a time, satisfying:*

   (a) **Segment check:** *Given $v \in \mathcal{V}$, Π can determine whether v is the first segment of some $X \in \mathcal{X}$.*

   (b) **Sequential extension and termination:** *Given a valid partial sequence*

   $$X|_k = (v_1, \ldots, v_k),$$

   Π *can determine either*

   - *the next segment v_{k+1} if the sequence is not yet complete, or*
   - *that v_k is the last segment of the sequence X.*

Proposition 6.1 (Computable Isomorphism of Assemblies). *Let $A = (\mathcal{V}, \Pi, \mathcal{X})$ be a sequentially constructible assembling set, and let E be a computable function enumerating $\mathcal{X}$. Then:*

1. *The concatenation operation Π can be implemented as a computable function.*

2. *For any other sequentially constructible assembling set $A' = (\mathcal{V}', \Pi', \mathcal{X}')$ with a computable enumerator E', if corresponding assembled strings $X \in \mathcal{X}$ and $X' \in \mathcal{X}'$ satisfy*

$$E(X) = E'(X') \implies W(X) = W'(X'),$$

then A and A' are isomorphic: there exists a bijection between the segments of $\mathcal{V}$ and $\mathcal{V}'$ that preserves the sequential structure of assemblies.

Proof (Part 1: Computable Construction of Π). Let $\Pi = (\pi_1, \pi_2, \pi_3, \pi_4)$ be defined as follows:

1. π_1: Given a segment $v \in \mathcal{V}$, determine whether v is the first segment of some assembled string $X \in \mathcal{X}$.

2. π_2: Given a partial assembly $X|_k = (v_1, \ldots, v_k)$, determine the next segment v_{k+1}.

3. $\pi_3 : \mathbb{N} \to \mathbb{N}$ is any total computable function that encodes the length $W(X_n)$ of the n-th assembled string.

4. π_4: An enumeration of assembled strings $X_n \in \mathcal{X}$.

Construction Algorithm:

Partition $\mathbb{N}$ into consecutive blocks corresponding to the segments of each X_n as follows:

1. Initialize $m = 0$.

2. For $n = 0, 1, 2, \ldots$:

   (a) Compute $\pi_3(n)$, which gives the length $W(X_n)$.

   (b) Assign consecutive numbers $m + 1, m + 2, \ldots, m + \pi_3(n)$ to be the segments $v_1, \ldots, v_{W(X_n)}$ of X_n.

   (c) Set $m = m + \pi_3(n)$ and continue.

This algorithm provides a *computable construction of* Π that satisfies all properties of a sequentially constructible assembling set: disjointness, completeness, and sequential assembly. The assembly set constructed by the partition of $\mathbb{N}$, using the above algorithm, is called the **fundamental assembly set**.

$\square$

Proof (Part 2: Isomorphism of Assemblies). Let $A = (\mathcal{V}, \Pi, \mathcal{X})$ be the **fundamental assembly set** constructed in Part 1. Let $A' = (\mathcal{V}', \Pi', \mathcal{X}')$ be any other sequentially constructible assembling set such that the enumerated strings $X_n \in \mathcal{X}$ and $X'_n \in \mathcal{X}'$ satisfy

$$W(X_n) = W(X'_n) \quad \text{for all } n.$$

Then A and A' are isomorphic.

Construction of the Isomorphism:

For each n, let the segments of X_n be

$$X_n = (v_1, \ldots, v_{W(X_n)}), \quad v_i \in \mathcal{V},$$

and let the segments of X'_n be

$$X'_n = (v'_1, \ldots, v'_{W(X'_n)}), \quad v'_i \in \mathcal{V}'.$$

Since $W(X_n) = W(X'_n)$, define a bijection

$$f_n : \{v_1, \ldots, v_{W(X_n)}\} \to \{v'_1, \ldots, v'_{W(X'_n)}\}$$

by mapping $v_i \mapsto v'_i$ for each i.

Combining all the f_n for all n defines a bijection

$$f : \mathcal{V} \to \mathcal{V}'.$$

$\square$

Let $X \in \mathcal{X}$ and let $X|_k = (v_1, \ldots, v_k)$ be any partial assembly of X. Then Π can sequentially generate $\mathcal{X}$ in the enumerated order until the full $X|_k$ is matched. This ensures that Π serves its intended purpose.

For the fundamental assembly set, this procedure is guaranteed to succeed even when the input $Y \notin \mathcal{X}$, in which case it is rejected. By the isomorphism property, it follows that all such assembly sets equipped with a function Π can similarly reject invalid inputs.

Definition 6.2 (Random Assembler). *Let $\mathcal{A} = (A, R_\mathcal{V})$ be a Random Assembler, where*

- $A = (\mathcal{V}, \Pi, \mathcal{X})$ *is an assembling set,*

- $R_\mathcal{V}$ *is a random generator producing elements of $\mathcal{V}$ according to some probability distribution $P_\mathcal{V}$.*

The Random Assembler $\mathcal{A}$ generates strings in $\mathcal{X}$ according to the following algorithm:

1. *Draw segments from $R_\mathcal{V}$ until a segment v_1 is obtained that is the first segment of some string $X \in \mathcal{X}$, as determined by Π.*

2. *Let $X|_k = (v_1, \ldots, v_k)$ be the current partial assembly.*

3. *At each step $k+1$, draw from $R_\mathcal{V}$ until a segment v is obtained such that $v = v_{k+1}$ according to Π and the current partial assembly $X|_k$.*

4. *If a drawn segment does not match v_{k+1}, reject it and redraw.*

5. *Repeat until the full string X is successfully assembled.*

This procedure defines a probability distribution $P_\mathcal{X}$ over $\mathcal{X}$ induced by the random generator $R_\mathcal{V}$ and the sequential assembly procedure Π.

Definition 6.3. *The* input probability $P_{\mathcal{V}}(X)$ *of* $X \in \mathcal{X}$ *with respect to* $R_{\mathcal{V}}$ *is defined as the probability that a segment* $v \in \mathcal{V}$ *generated by* $R_{\mathcal{V}}$ *belongs to the construction* $X = v_1 \dots v_{W(X)}$, *i.e.,*

$$P_{\mathcal{V}}(X) = \sum_{v \in \mathcal{V}} P_{\mathcal{V}}(X \mid v) \, R_{\mathcal{V}}(v),$$

Proposition 6.2 (A power law for the input probability $P_{\mathcal{V}}(X)$)**.** *Let* $\langle W \rangle_{\mathcal{V}} = w$ *be the expected segment count, and let*

$$H_{\mathcal{V}}(X) = - \sum_{X \in \mathcal{X}} P_{\mathcal{V}}(X) \ln P_{\mathcal{V}}(X)$$

be the entropy over strings in $\mathcal{X}$. *The distribution* $P_{\mathcal{V}}(X)$ *that maximizes* $H_{\mathcal{V}}$ *under the constraint* $\langle W \rangle_{\mathcal{V}} = w$ *is*

$$P_{\mathcal{V}}(X) = \frac{W(X)^{-\alpha}}{Z(\alpha)}, \qquad Z(\alpha) = \sum_{X \in \mathcal{X}} W(X)^{-\alpha}.$$

Equivalently, $Z(\alpha)$ *can be expressed as the*

$$Z(\alpha) = \int W^{-\alpha} g(W) \, dW,$$

where $g(W)$ *is the density of strings with weight* W, *and is analogous to a **Mellin Transform**.*

Proposition 6.3 (An input power law from fixed information rate)**.** *Let the* information rate *with respect to the segment distribution* $R_{\mathcal{V}}$ *be defined as*

$$I_{\mathcal{V}} = \lim_{N \to \infty} \frac{H_{\mathcal{V}^N}}{N} = H(P_{\mathcal{V}}) + \mathbb{E}_{P_{\mathcal{V}}}[\ln W(X)],$$

where

$$H(P_{\mathcal{V}}) = - \sum_{X \in \mathcal{X}} P_{\mathcal{V}}(X) \ln P_{\mathcal{V}}(X),$$

and where the conditional probability $P_{\mathcal{V}}(v \mid X)$ *is uniform over the segments of* X.

Fix the information rate $I_{\mathcal{V}} = I_0$ *as a constant constraint. Then the distribution* $P_{\mathcal{V}}(X)$ *that maximizes the entropy* $H(P_{\mathcal{V}})$ *under this constraint is satisfied by*

$$P_{\mathcal{V}}(X) = \frac{W(X)^{-\alpha}}{Z(\alpha)}, \qquad Z(\alpha) = \sum_{X \in \mathcal{X}} W(X)^{-\alpha}.$$

for some α. *Equivalently, the partition function can be written as*

$$Z(\alpha) = \int W^{-\alpha} g(W) \, dW,$$

where $g(W)$ *denotes the density of strings with weight* W. *This representation is analogous to a **Mellin transform**.*

Proposition 6.4 (Power law for the Output Probability $P_{\mathcal{X}}(X)$)**.** *Let* $X \in \mathcal{X}$ *and assign a uniform conditional distribution to its segments:*

$$P_{\mathcal{V}}(v \mid X) = \frac{1}{W(X)} \quad \text{for all } v \text{ in construction of } X.$$

Assume the input probability over strings is

$$P_{\mathcal{V}}(X) = \frac{W(X)^{-\alpha}}{Z(\alpha)}, \qquad Z(\alpha) = \sum_{X \in \mathcal{X}} W(X)^{-\alpha}.$$

Then, the output probability of X *under the Random Assembler* $\mathcal{A}$ *is*

$$P_{\mathcal{X}}(X) = \frac{W(X)^{-(\alpha+1)}}{Z(\alpha+1)}.$$

Proof. If the random generator $R_\mathcal{V}$ generates the first segment v_1 of some string $X \in \mathcal{X}$, then X is deterministically constructed in sequence as

$$X = v_1 \ldots v_{W(X)}$$

by the action of Π in the Random Assembler $\mathcal{A}$.

Under the uniform conditional assumption,

$$P_\mathcal{V}(v \mid X) = \frac{1}{W(X)} \quad \text{for all } v \in X,$$

and the input distribution

$$P_\mathcal{V}(X) = \frac{W(X)^{-\alpha}}{Z(\alpha)},$$

the quantity of interest is the conditional probability that a generated first segment corresponds to the string X, namely

$$P_\mathcal{X}(X) = P_\mathcal{V}(X \mid 1).$$

By Bayes' theorem,

$$P_\mathcal{X}(X) = \frac{P_\mathcal{V}(1 \mid X)\, P_\mathcal{V}(X)}{P_\mathcal{V}(1)} = \frac{1}{W(X)} \cdot \frac{W(X)^{-\alpha}}{Z(\alpha)} \cdot \frac{1}{P_\mathcal{V}(1)}.$$

The normalization term satisfies

$$P_\mathcal{V}(1) = \sum_{X \in \mathcal{X}} P_\mathcal{V}(v_1 \mid X)\, P_\mathcal{V}(X) = \sum_{X \in \mathcal{X}} \frac{1}{W(X)} \cdot \frac{W(X)^{-\alpha}}{Z(\alpha)} = \frac{Z(\alpha+1)}{Z(\alpha)}.$$

Substituting yields

$$P_\mathcal{X}(X) = \frac{W(X)^{-(\alpha+1)}}{Z(\alpha+1)}.$$

$\square$

Definition 6.4 (Semantic Mapping onto a Turing Machine Language). *Let $A = (\mathcal{V}, \Pi, \mathcal{X})$ be a sequentially constructible assembling set. A semantic mapping is a computable bijection*

$$\Xi : \mathcal{X} \to L(M)$$

onto the language $L(M) \subseteq \{0,1\}^*$ of a Turing machine M.

Intuitively, A specifies how *sequences are assembled, while Ξ assigns to each assembled sequence a* meaning *in the form of a string accepted by M.*

Definition 6.5 (Canonical Assembling Set). *Let $A = (\mathcal{V}, \Pi, \mathcal{X})$ be an assembling set. It is called canonical if, for each $X \in \mathcal{X}$, there exists a unique $v_1 \in \mathcal{V}$ such that*

$$X = \Pi(v_1), \qquad |v_1| = -\log P_\mathcal{X}(X),$$

where $P_\mathcal{X}(X)$ is the probability of assembling X.

Proposition 6.5 (Computable and Prefix-Free Semantic Mapping). *Let $A = (\mathcal{V}, \Pi, \mathcal{X})$ be a canonical assembling set, and let M' be a Turing machine with language $L(M') \subseteq \{0,1\}^*$. Then there exists a semantic mapping*

$$\Xi : \mathcal{X} \to L(M')$$

that is computable *and bijective, and there exist a prefix-free Turing machine $M = (A, \Xi)$.*

Proof. Since both $\mathcal{X}$, and $L(M)$ are enumerable, there exist a bijection. Since $\sum_X P_\mathcal{X}(X) = 1$, the Kraft–McMillan theorem guarantees the existence of a prefix-free code with lengths

$$|v_1| = \lceil -\log P_\mathcal{X}(X) \rceil.$$

Enumerate $\mathcal{X}$ and assign to each $X \in \mathcal{X}$ its corresponding prefix-free code. This defines a computable, injective mapping $\Xi : \mathcal{X} \to L(M)$ whose outputs form a prefix-free set. $\qquad\square$

With reference to **Proposition 3.13**, let M be a prefix-free machine corresponding to a canonical assembling set $A = (\mathcal{V}, \Pi, \mathcal{X})$, and let Ξ be an appropriate computable semantic mapping. Relative to the universal prefix-free machine U, the following inequality holds:

$$\max_{y \in L(U)} \left\{ K_U(y) - \left(-\log P_\mathcal{X}(X) \Big|_{\substack{M(v_1)=y \\ \Pi(v_1)=X}} \right) \right\} \leq C_U(M) \tag{6.1}$$

where $P_\mathcal{X}(X)$ is the assembly probability of $X \in \mathcal{X}$, and $-\log P_\mathcal{X}(X) \Big|_{\substack{M(v_1)=y \\ \Pi(v_1)=X}}$ is the **probabilistic-complexity** and $C_U(M)$ is the **explanatory-complexity**.
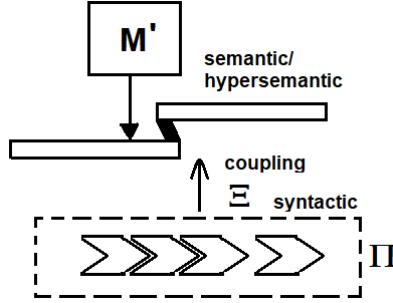
Figure 17: Computational view of Jigsaw-puzzle model: Π *specifies the syntactic rules for the assembling set A. The Turing machine M' assigns semantic meaning to elements $X \in \mathcal{X}$ within A. Ξ serves as the coupling mechanism linking the language $L(M')$ with $\mathcal{X}$.*

Generalized computational theory does not require strict adherence to Turing computability. For instance, Ξ could correspond to a function that is uncomputable in the Turing sense. This perspective highlights a resemblance to certain emergent phenomena, in which each individual component contains information that cannot be fully described in isolation, yet when integrated through an appropriate mechanism, they form a coherent whole, analogous to a real game of jigsaw puzzle.

6.1 Applications to the Phenomenon of Life- Continued...

The preceding discussion illustrated that strings considered highly improbable in Chaitin's framework may correspond to more probable outcomes when a more complex explanatory mechanism is taken into account, establishing a fundamental tradeoff between probability and explanatory complexity implied by **Proposition 3.13**.

A similar principle can be applied to life: while the probability of a phenomenon as complex as life emerging may appear extremely low, considering it as the result of molecular interactions increases its likelihood, yet the manner in which these interactions give rise to higher-level functional structures, such as protein and RNA machines, remains difficult to explain. This mirrors the **probability–explanatory complexity tradeoff** as in **equation 6.1**.

According to a **generalized computational framework**, it is conceptually valid to postulate a hypothetical machine, called the **generalized computer**, which takes as input a description of a phenomenon

and internally reconstructs it, subsequently producing an output description of the same phenomenon that is built into the machine itself. This allows a phenomenon to be explained from a concise key description through a detailed algorithm. For instance, one could feed an algorithm for constructing a car into this machine; it would internally assemble the car according to the algorithm and then output a description of the constructed object, for example by taking a photograph of the car and encoding it as a string. The generalized computer possess internal oracles capable of understanding and executing instructions encoded in the input description. Similarly, one could hypothetically input the mechanism by which a bacterium forms, starting from basic molecules; if the process succeeds, the output might, for example, be a detailed representation or image of a bacterium.

The complexity of a phenomenon can be defined as the minimum length of the description input to a hypothetical generalized computer required to recreate it, giving rise to the notion of a **generalized Kolmogorov complexity**. Of course, this measure depends on the specific design of the generalized computer, and complexity is typically evaluated relative to a machine possessing certain well-defined properties. While the preceding discussion focused on Turing machines, the same principles apply naturally within the broader framework of generalized computational theory. Viewed in this light, as with Turing machines, the complexity of a description can be decomposed into **probabilistic-complexity** and **explanatory-complexity**, allowing a higher probability to be achieved at the cost of a more elaborate explanatory mechanism.

What has been accomplished is the following: if the universe is conceived as a computational system, then life can be viewed as one of its programs, particularly analogous to a virus. In this perspective, the universe itself possesses a description akin to the code of a computer, and the description of every phenomenon within it can be seen as analogous to computer code for programs. Specifically, the description of life resembles the code for an evolving virus, capturing the dynamics and adaptive processes of living systems.
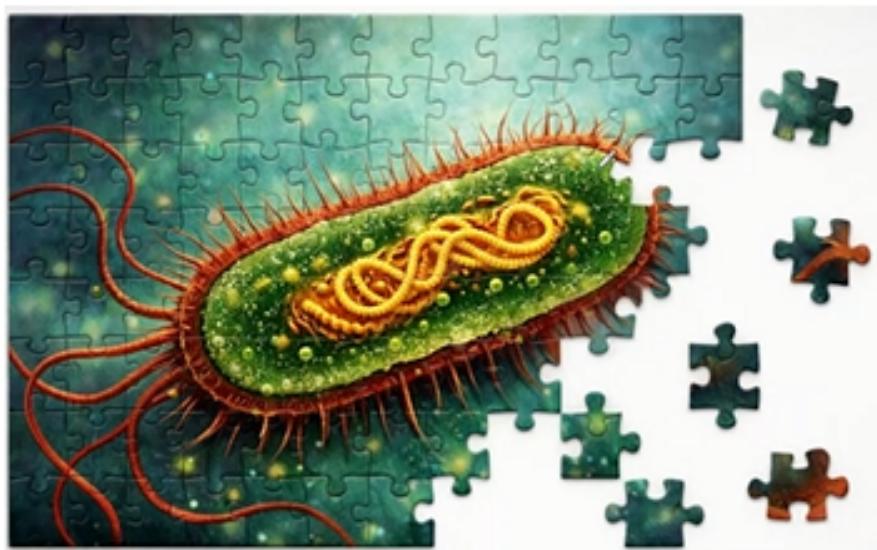

Figure 18: An illustrative representation of life conceptualized as a jigsaw puzzle.

7   On Thermodynamic Phenomenon

Physics of simple systems is often formulated in terms of explicit equations of motion. As systems become large, heterogeneous, or strongly interacting with their surroundings, a thermodynamic description becomes more informative. In this viewpoint, the system is treated as an open system exchanging energy, matter, or information with a larger environment.

Classical equilibrium thermodynamics is largely silent about dynamics, since it characterizes systems only in equilibrium states and through state functions. By contrast, **non-equilibrium thermodynamics** explicitly concerns the dynamics of systems away from equilibrium, where time-dependent processes such as transport, dissipation, and relaxation become essential.

A system out of equilibrium is characterized by the presence of **thermodynamic gradients**—such as gradients in temperature, chemical potential, or pressure—which drive fluxes and irreversible processes. These gradients sustain currents and enable structured behavior that cannot exist at equilibrium.

It has become increasingly apparent that the use of **gates** in thermodynamics—partly inspired by concepts from quantum information theory—provides a particularly effective framework for describing dynamics in complex thermodynamic systems. This perspective has been especially fruitful in the thermodynamics of computation, where logical operations are modeled as physical processes that regulate the flow of energy, entropy, and information.
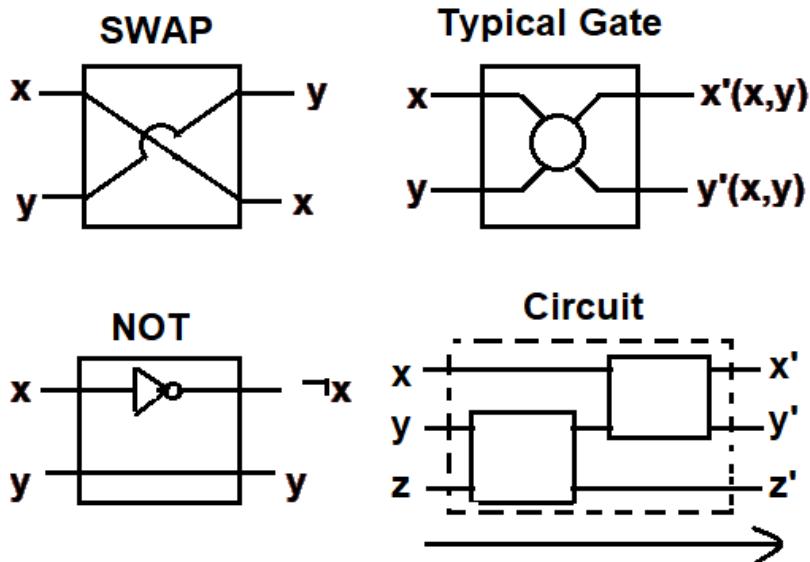

Figure 19: Thermodynamic Gates: *Examples of gates relevant for thermodynamic systems.* ***Top left:*** *the SWAP gate, which exchanges two input states; it conserves entropy and does not generate correlations by mixing different inputs through a global interaction, making it particularly useful in many applications.* ***Bottom left:*** *the simplest local interaction, the NOT gate, which flips a 0 to 1 and a 1 to 0.* ***Top right:*** *a typical two-input gate that generates correlations via global interactions between its inputs.* ***Bottom right:*** *a composite gate formed from multiple two-input gates, illustrating an internal interaction structure.*

Thermodynamic gates are understood operationally as transformations applied to states propagating along channels. Each channel carries a system state—such as a bit, particle occupation number, or local energy level—while the gate implements a rule that modifies the state according to its internal interaction structure.

A complex thermodynamic system can often be decomposed into constituent subsystems, each carrying its own degrees of freedom. In the channel–gate framework, each subsystem is assigned a unique **channel** that encode its current state, such as a binary state, energy level, or occupation number.

Interactions between subsystems are then represented as *gate operations* acting on the channels corresponding to the interacting subsystems. Formally:

In this framework, one imagines a network of channels along which states evolve in discrete time steps. At each step, gates act on one or more channels, updating the state of the system according to the gate's function.

It may be surprising, but SWAP gates are remarkably useful in thermodynamic contexts. Consider, for example, a gas partitioned by a wall. The transfer of microstates—or the redistribution of entropy—from one side to the other, without changing the total entropy of the system, can be viewed as a simple SWAP-like operation.
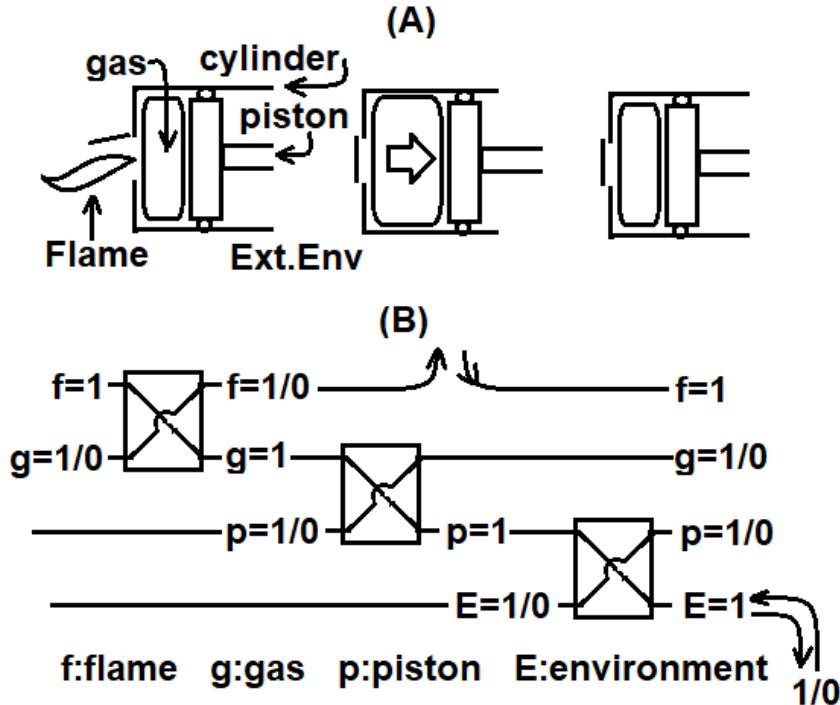

Figure 20: Engine thermodynamics with gates: **(A)** *Typical operation cycle of a thermodynamic engine, as conventionally discussed. A flame is introduced to the gas in the cylinder, causing it to expand and push the piston. Work is performed on the environment, and energy initially supplied by the flame is dissipated. The cycle then returns to the initial state where the flame is introduced, completing one engine cycle.* **(B)** *Representation of information flow in the same system using gates and channels. The flame, gas, piston, and environment are each assigned their own channel. SWAP gates are arranged to transfer information from the flame to the environment in a reversible manner. In particular, the SWAP operation can be represented as $(1/0, 1) \rightarrow (1, 1/0)$, where $1/0$ denotes a maximally uncertain entropic state. This illustrates how entropy and information flow can be tracked explicitly using a gate–channel framework.*

In a typical binary gate, let X and Y denote the input channels, and for simplicity, let their respective states also be denoted by X and Y, avoiding unnecessary symbols. The output of the gate is then represented by X' and Y', which are functions of the inputs:

$$X' = X'(X, Y), \qquad Y' = Y'(X, Y),$$

where X' and Y' refer to the updated states of the input channels X and Y, respectively.

It is natural to assume that the joint entropy is conserved through the gate:

$$H(X, Y) = H(X', Y'),$$

reflecting the fact that the gate implements a reversible transformation. This assumption is consistent with a fine-grained description of the system in the Hamiltonian framework, where microscopic dynamics are entropy-preserving.

For the SWAP gate, the entropy-conservation condition is naturally satisfied. In fact, it is the only simple global gate operation for which this holds automatically.

It may be confusing what the 1s and 0s represent in this context. To clarify, information is typically measured in *bits*, which provide a minimal unit of information. A **bit** is a system that can exist in one of two distinguishable states, commonly labeled 0 and 1. It can be thought of as the simplest carrier of information, representing the outcome of a binary choice or the state of a two-level system.

The values 1 and 0 can be understood as the answers to a specific question. Without the context of the question, these symbols have no intrinsic meaning. For example, a particular channel X may carry a 1 or a 0 in response to a question about X, while another channel Y may also carry a 1 or a 0, but corresponding to a different question. In this sense, the meaning of a bit is relative to the context of the system and the specific property it encodes.

In quantum information theory, there is a well-known **no-cloning theorem**, which states that it is impossible to create two identical copies of an arbitrary unknown state. A closely related limitation also appears in thermodynamics when one imposes reversibility at the fine-grained level. Under this constraint, information cannot be copied into a noisy environment in a way that removes or displaces the noise. Instead, the noise must either be distributed across the resulting copies or be transferred into the copying mechanism itself.

This limitation can be viewed as a thermodynamic analogue of the no-cloning principle. Any reversible copying process necessarily preserves total entropy, implying that noise cannot be eliminated but only redistributed. For example, a copier constructed from a collection of SWAP gates applied locally—such as one gate per pixel—acts by mixing a noisy system with a less noisy one. As a result, some noise flows from the noisy system into the initially clean system, while some information from the clean system flows into the noisy system. The copying process therefore exchanges information and noise rather than creating a noise-free duplicate.
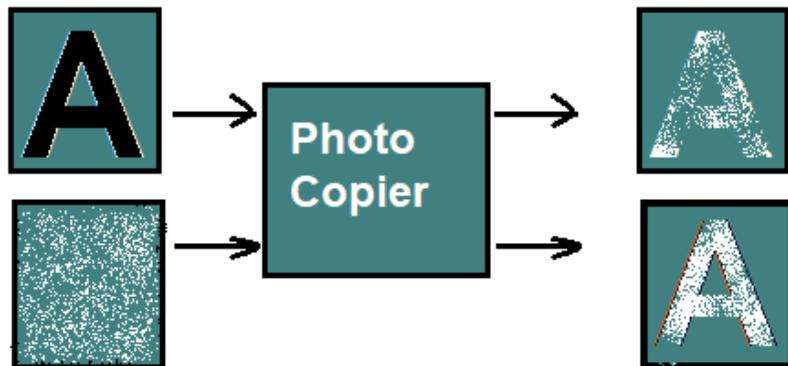

Figure 21: No Cloning Theorem in Thermodynamics: *Under information-conserving dynamics, it is not possible to copy information into a noisy environment in a way that eliminates the noise. Any attempt at copying must either distribute the noise across the resulting copies or transfer the noise into the copying mechanism itself.*

Under information conservation, information behaves in a manner closely analogous to energy: it flows from one channel to another and cannot be created or destroyed. This principle can be observed in both man-made and biological machines. Systems that produce more ordered outputs from more disordered inputs typically exhibit an accumulation of thermal or structural damage after repeated operation cycles.

For example, heat engines convert chaotic, high-entropy thermal motion into directed, ordered work, but in doing so they inevitably generate waste heat and experience heating themselves. Similarly, biological

systems—such as proteins that catalyze the synthesis of complex molecules—operate by channeling disordered molecular interactions into highly structured outcomes, while dissipating energy and entropy into their surroundings. In both cases, the emergence of order is accompanied by the redistribution rather than the elimination of disorder, reflecting the fundamental constraints imposed by information conservation.

Of course, many systems incorporate mechanisms to mitigate or repair thermal damage. In engines, coolant systems remove excess heat by exchanging energy, entropy, and information between the heated engine and a cooler reservoir. In biological systems, specialized proteins repair damaged molecular structures, while irreparably damaged components are degraded and replaced through metabolic processes.

However, these repair mechanisms are themselves subject to the same thermodynamic constraints. If the entropy generated during repair is not dissipated into an external environment, the repair mechanism will accumulate damage of its own.

The following model formalizes these ideas by describing information transfer in machines.

7.1 Modeling Information Transfer in Machines

This model describes a recyclable machine M acting between a system X and an environment E. The model consists of three channels, denoted by E, M, and X. Here, E represents the environment, M the machine, and X the system on which the machine acts.

At the beginning of an operational cycle, at time $t = 0$, the machine and the system are assumed to be statistically independent:
$$H(M, X)\big|_{t=0} = H(M) + H(X).$$

The machine then acts on the system through a gate operation,
$$(M, X)_{t=0} \;\longrightarrow\; (M', X')_{t=1},$$
where
$$X' = f(X), \qquad M' = M'(M, X).$$
The transformation f reduces the entropy of X, producing a more ordered system. As a consequence of this entropy reduction, the machine transitions to a degraded state M', reflecting thermal or structural damage accumulated during the operation.

The environment E contains the information necessary to restore the machine to its original state. This repair process is modeled by a second gate operation,
$$(E, M')_{t=1} \;\longrightarrow\; (E', M)_{t=2}.$$

During this interaction, information is transferred from the environment to the machine, restoring $M' \to M$. As a result, the environment transitions to a higher-entropy state E', having lost the information required for repair.

Thus, the overall cycle demonstrates that entropy reduction in the system X and repair of the machine M are achieved not by eliminating disorder, but by exporting it to the environment. **Figure 22** illustrates this information flow in greater detail.

The calculations show the following. Under information conservation and reversibility, $H(M, X)|_{t=0} = H(M', X')|_{t=1}$ and $H(E, M')|_{t=1} = H(E', M)|_{t=2}$. Under determinism, $H(M' \mid M, X) = 0$ and $H(X' \mid X) = 0$, and since the machine is restored at the end of the cycle, $\Delta_{20} H(M) = 0$.

Because the machine and the system are initially independent, $H(M, X)_{t=0} = H(M)_{t=0} + H(X)_{t=0}$, and after interaction, $H(M', X')_{t=1} = H(M')_{t=1} + H(X')_{t=1} - I(M'; X')|_{t=1}$.

For the system, $H(X, X') = H(X) + H(X' \mid X) = H(X)$, since $H(X' \mid X) = 0$. Similarly, $H(X', X) = H(X') + H(X \mid X')$, and since $H(X, X') = H(X)$, it follows that $H(X') = H(X) - H(X \mid X') \leq H(X)$. Therefore, $\Delta_{20} H(X) = \Delta_{10} H(X) = -H(X \mid X')$, representing a reduction in entropy.

Furthermore, $H(M')_{t=1} = H(M,X)_{t=0} + H(M' \mid M,X) = H(M) + H(X)$, since $H(M' \mid M,X) = 0$. This demonstrates the thermal damage to the machine, $\Delta_{01}H(M') = H(X) \geq 0$.

Because the environment and the degraded machine are uncorrelated at $t = 1$, $H(M',E)_{t=1} = H(M')_{t=1} + H(E)_{t=1}$, and since $H(M')_{t=1} = H(M) + H(X)$, it follows that $\Delta_{20}H(E) = \Delta_{21}H(E) = H(M) + H(X)$.

By reversibility, the evolution

$$(E,(M,X)) \longleftrightarrow (E,(M',X')) \longleftrightarrow ((E,M'),X') \longleftrightarrow ((E',M),X')$$

preserves the total information content, and therefore

$$H(E,M,X)\big|_{t=0} = H(E',M,X')\big|_{t=2},$$

as expected.
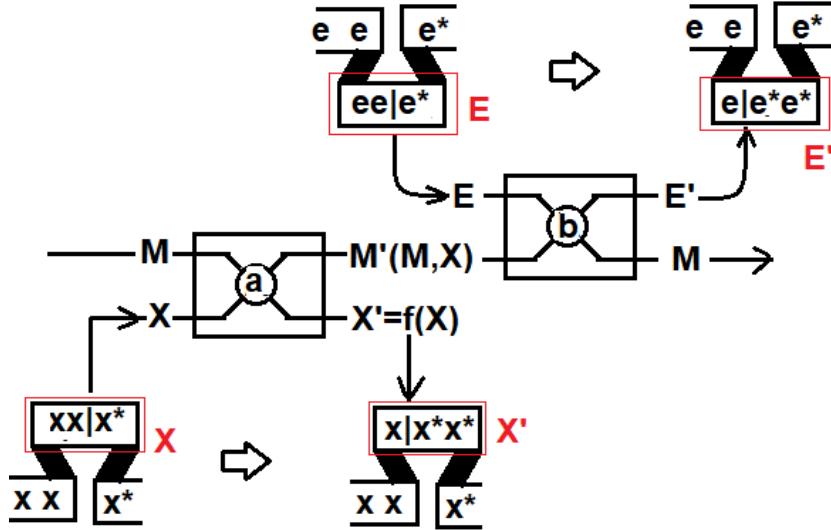
Figure 22: Model for information transfer in machines: *This model describes a recyclable machine M acting between a system X and an environment E. A gate-based analogy combined with computational machines that use tapes provides an elegant model for explaining the behavior of machines in a simplified setting. The environment E and the system X are modeled as moving locations on separate tapes (indicated by red box). The tape for E is of the form $\dots ee \mid e^* \dots$, and the tape for X is of the form $\dots xx \mid x^* \dots$. Through interaction with the machine M and its degraded form M', these configurations undergo transformations of the form $\dots ee \mid e^* \dots \to \dots e \mid e^*e^* \dots$ and $\dots xx \mid x^* \dots \to \dots x \mid x^*x^* \dots$, respectively. Here, x and e denote the states of X and E prior to interaction, while x^* and e^* symbolically represent their transformed states.*

Finally, because $H(E,M,X)|_{t=0} = H(E',M,X')|_{t=2}$, and because $H(E,M,X)|_{t=0} = H(E)|_{t=0} + H(M)|_{t=0} + H(X)|_{t=0}$, while $H(E',M,X')|_{t=2} = H(E')|_{t=2} + H(X')|_{t=2} + H(M)|_{t=2} - I(E';X') - I(M;E',X')$, it follows that the entropy created in the environment over the full cycle is given by

$$\Delta_{20}H(E) = -\Delta_{20}H(X) + I(E';X') + I(M;E',X').$$

The entropy increase, or equivalently the information lost from the environment E, is accounted for by the information gained by the system X, together with the mutual information terms: $I(E';X')$ describing correlations generated between E and X, and the memory of the machine acting on X and E, quantified by $I(M;E',X')$.

7.2 Thermodynamic Fate of Machines

The previous section modeled a machine M acting between an environment E and a system X, increasing the entropy of the environment while decreasing that of the system in a cyclic manner. During each cycle, M returns to its original state, while E and X are transformed into E' and X'. The system X interacts with M to produce X' and a degraded machine state M', while the environment E subsequently interacts with M' to restore it to M, with the environment itself being transformed into E' in the process.

The information required to repair the degraded machine M' is absorbed from the environment E, resulting in a corresponding degradation of E to E'. Since the available information in the environment is finite(This can, for example, be modeled by considering the tape corresponding to E in **Figure 22** to be finite.), only a finite number of such cycles can occur. Once this information is depleted, the machine becomes trapped in a degraded state and can no longer be restored to M, potentially undergoing further degradation. This appears to be the typical fate of machines, both man-made and naturally occurring, including biological systems.
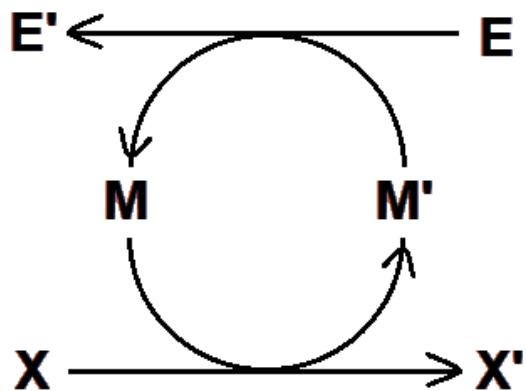

Figure 23: cycle diagram for (E, M, X): *A cyclic interaction diagram illustrating the flow of information and entropy between the system X, the environment E, and the machine M. The machine undergoes degradation $(M \rightarrow M')$ while acting on X, and is subsequently restored through interaction with E, which is transformed to E'.*

Of course, one can propose the existence of another machine R, a **recoverer**, that maps $E' \rightarrow E$. However, this would require an additional information source. Theoretically, it is possible for $(E, X) \rightarrow (E', X')$ and then back $(E', X') \rightarrow (E, X)$. Yet, the two paths require two different algorithms, and therefore two different machines, typically of distinct complexity. For instance, given two prime numbers p and q, computing their product $N = p \cdot q$ is straightforward, whereas recovering p and q from N through factorization is computationally difficult. A mismatch in computational difficulty itself can render one path significantly less probable than the other. As a consequence, most cyclic processes are not expected to be permanent.

Life, for instance, is composed of many metabolic cycles that appear to be permanent within a closed system. However, these cycles depend on external gradients, such as the temperature difference between the Sun and the Earth, and the geothermal gradient between the Earth's interior and its surface, to sustain them. Mars, for example, has lost its geothermal gradient, and as a result, life is no longer expected to exist there.

The traditional view holds that entropy increases according to some version of the second law, and that systems which dissipate or produce entropy more rapidly tend to dominate. This perspective, however, makes little sense at a fundamental level. It is more likely that what actually occurs is that some systems are able to extract information from their environment more efficiently, which makes them dominant because they acquire the information necessary to maintain themselves and grow at the expense of others. As a result, the environment accumulates entropy at a comparable rate.
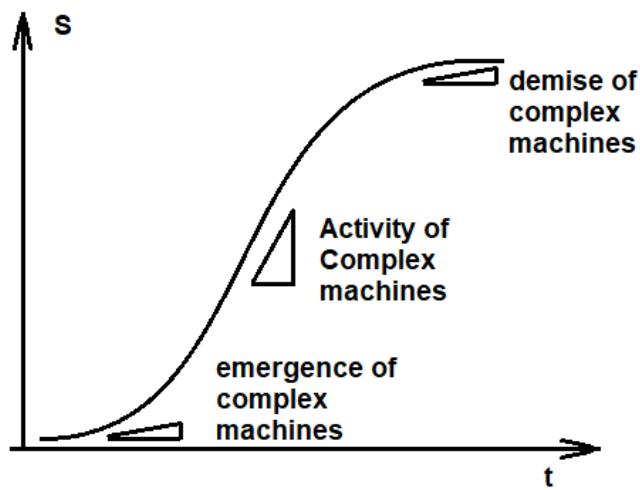
Figure 24: Thermodynamic Fate of Machines: *The diagram depicts how the evolution of machines corresponds to entropy production in the environment. Initially, at a low entropic level, sufficient information is available for the self-assembly of simple machines. Because these machines can extract information from the environment only at a slow rate, they increase environmental entropy relatively slowly. Over time, simple machines extract information and evolve into more sophisticated and competitive machines, which are capable of extracting information at higher rates and consequently increasing environmental entropy more rapidly. As this process continues, it accelerates approximately exponentially. However, the machines are unable to recover all of the information they deplete. As a result, a point is eventually reached at which insufficient information remains in the environment to sustain the most sophisticated machines. These machines then degrade into progressively simpler forms that require lower rates of information extraction. Correspondingly, the rate of entropy increase in the environment declines, until ultimately all machines are destroyed once the remaining unrecoverable information has been exhausted.*

8 Bibilography

[**A1**] Mutschler, H., Taylor, A. I., Porebski, B. T., Lightowlers, A., Houlihan, G., Abramov, M., Herdewijn, P., & Holliger, P. (2018). *Random-sequence genetic oligomer pools display an innate potential for ligation and recombination.* eLife, 43022. `https://doi.org/10.7554/eLife.43022`

[**A2**] Root-Bernstein, R., Baker, A. G., Rhinesmith, T., Turke, M., Huber, J., & Brown, A. W. (2023). *"Sea Water" Supplemented with Calcium Phosphate and Magnesium Sulfate in a Long-Term Miller-Type Experiment Yields Sugars, Nucleic Acids Bases, Nucleosides, Lipids, Amino Acids, and Oligopeptides.* Life, 13(2), 265. `https://doi.org/10.3390/life13020265`

[**A3**] Criado-Reyes, J., Bizzarri, B. M., García-Ruiz, J. M., Saladino, R., & Di Mauro, E. (2021). *The role of borosilicate glass in Miller–Urey experiment.* Scientific Reports, 11, 235. `https://doi.org/10.1038/s41598-021-00235-4`

[**B1**] Murawski, R. (1999). *Recursive Functions and Metamathematics: Problems of Completeness and Decidability, Gödel's Theorems.* Springer.

[**B2**] Allison, C. D. (2021). *Foundations of Computing: An Accessible Introduction to Formal Languages.* Fresh Sources, Inc.

[**B3**] Filiol, É. (2005). *Computer Viruses: From Theory to Applications.* Springer.

[**B4**] Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.).

[**B5**] Fortnow, L. (*n.d.*). *Kolmogorov Complexity.*

[**B6**] Tadaki, K. (2019). *A Statistical Mechanical Interpretation of Algorithmic Information Theory.* Springer Singapore.

[**B7**] Syropoulos, A. (2008). *Hypercomputation: Computing Beyond the Church-Turing Barrier.* Springer.

[**B8**] Draeger, J. (*n.d.*). *Quantum Turing Machines.* Contribution to the Encyclopedia of Mathematics.

[**B9**] Adleman, L. M. (*n.d.*). *An Abstract Theory of Computer Viruses.* Department of Computer Science, University of Southern California.

[**C1**] *Automatic Mechanical Self Replication.* (n.d.). [Video]. YouTube. Retrieved from https://www.youtube.com/watch

[**C2**] Freitas Jr., R. A., & Merkle, R. C. (2004). *Kinematic Self-Replicating Machines.* Landes Bioscience / Eurekah.com.

[**C3**] von Neumann, J. (2002). *Theory of Self-Reproducing Automata.* UMI Reprint, University of Illinois, 1966 Edition.

[**D1**] Hathurusinghe. Y (2025). *On the phenomenon of consciousness*

[**E1**] How quantum mechanics help birds find their way. YouTube nature video channel. `https://www.youtube.com/watch?v=0SPD2r0xV8k`.

[**E2**] Peter Hore on Radical pair mechanism of magnetoreception. FENS channel. `https://www.youtube.com/watch?v=FytxLiHlah4`.

[**E3**] Mohseni, M., Rebentrost, P., Lloyd, S., & Aspuru-Guzik, A. (2008). Environment-assisted quantum walks in photosynthetic energy transfer. *The Journal of Chemical Physics, 129(17)*.

[**F1**] Endres, R. G. (2025). The unreasonable likelihood of being: origin of life, terraforming, and AI. *arXiv:2507.18545v1*.