1 Chapter 1: Self-Reference in Programs

Let us begin with a science-fiction thought experiment. Imagine you are planning to colonize Mars by deploying a fleet of machines designed to transform the planet's atmosphere. Very quickly, a problem becomes apparent: each machine has limited durability, and given the scale of the project, continuously shipping replacements from Earth would cause costs to skyrocket.

Suppose, however, that these machines are built from materials abundant on Mars. This suggests a way around the transportation problem: design an initial generation of machines that, in addition to contributing to the terraforming process, can produce exact copies of themselves. These copies would then continue the same task, allowing the system to sustain and expand itself without further shipments from Earth.

At first glance, it is not obvious that such a design is even possible. Intuitively, if a machine $\mathcal{A}$ is to build a machine $\mathcal{B}$, then $\mathcal{A}$ must contain complete information about $\mathcal{B}$, along with additional mechanisms required to construct $\mathcal{B}$ from that information. It seems to follow that $\mathcal{A}$ must be more complex than $\mathcal{B}$. This leads to an apparent paradox: how could $\mathcal{A}$ construct a machine of equal complexity? Yet, biology has clearly found a way to resolve this difficulty. A key insight, originally due to von Neumann, is to separate the system into two distinct components: the machine, which carries out physical operations according to instructions, and a blueprint containing those instructions. Specifically, the blueprint defines the machine's function in the terraforming process, as well as instructions for the machine to construct an exact copy of itself just before it goes out of order. For the self-replication to be truly complete, the new machine must also be equipped with the exact same blueprint to keep carrying out the same functionality, and eventually, to also self-replicate. Therefore, the blueprint must inevitably contain the instruction: "And now, copy these instructions." referring to its own description.

In this way, we decouple the problem into two parts: designing the self-referential content of the blueprint, and constructing a physical machine capable of executing it. We will address the second challenge in the next chapter, where we discuss the remarkable work of John von Neumann. For now, our focus will be entirely on the design of the blueprint itself. To that end, we give a brief introduction to the theory of computation, using the standard terminology introduced by Turing. We will examine the existence of quines—programs that output their own source code—distinguishing between universal constructions and weaker variants. Finally, we turn to Kleene's recursion theorems, which provide some of the deepest theoretical results on self-reference. These theorems offer a systematic method for constructing such self-referential blueprints in almost any programming language.

Remark 1.1. *This kind of thought experiment was seriously considered by some of the most influential scientists of the 20th century, most notably by John von Neumann whose work gave rise to the term von Neumann probe [9, 2]; we will discuss his work in detail in the next chapter.*

1.1 Turing Machines

Most readers have probably developed an intuition for what an algorithm or a program is. After all, one encounters this topic in any computer science class when learning Python, C++ or any other popular programming language. However, rarely does one need to think about the formal mathematical definition of a program. Yet, the need for such a definition became apparent from a series of fascinating events in the history of mathematics around the 1930s. For interested readers, we briefly summarize this in Section 1.1.1. For now, let us delve right into the definition of a Turing machine: a notion that offers a very intuitive and constructive definition of what a program is.

1.1.1 Historical Window: The Need to Define Algorithms

At the turn of the 20th century, mathematics had been shaken at its foundations. Cantor's work on set theory allowed constructing uncontrollably large objects such as ''the set of all sets'' leading to inconsistencies in mathematics such as the Russell's paradox:

$$\text{Let } R = \{S \text{ a set} \mid S \notin S\}, \text{ then } R \in R \iff R \notin R.$$

The need to resolve such problems and to put mathematics on stable foundations became apparent. David Hilbert proposed a program aiming exactly for this. It consisted of two main points:

1. Axiomatization and consistency of mathematics: List a set of axioms from which all mathematical truths would be provable. Specifically, prove the consistency of such an axiomatic system.

2. Entscheidungsproblem: Design an algorithm that would determine whether any given formula is provable from the axioms or not. (Here, a proof is a finite sequence of statements where the correctness of each step can be checked by an algorithm.)

The first aim was quickly crushed by Gödel and his incompleteness theorems, the first one stating that any reasonable (consistent, sufficiently expressive, enumerable by a program) axiomatic system is incomplete: there will be true statements that cannot be proven; the second theorem further shows that for any such axiomatic system its consistency cannot be proven within the system. For the second problem, in order to show that no algorithm solving this task exists, one has to first define precisely what an algorithm is. Different suggestions of how to formalize this came from Alonso Church and Alan Turing who independently refuted the second problem. Whereas Church's approach based on λ-calculus did not make it obvious that his formalism aligns with the general intuition of what an algorithmically solvable function is, Turing in his seminal work [8] provides a convincing discussion of why his ''computing machines'' capture this intuition well. Curiously, Church's lambda calculus and Turing's machines were proven to be equivalent notions - a sign they are characterizing something profound that indeed laid grounds to a new field.

Fix a finite set Σ. The set Σ^* denotes all finite sequences over Σ, including the empty sequence ε.

Definition 1.2 (Turing Machine)**.** *A Turing machine (TM) is a triple $\mathcal{T} = (Q, \Sigma, \delta)$ where:*

1. *Q is a finite set of states, with $q_0 \in Q$ the* initial state *and $q_{\text{halt}} \in Q$ the* halting state*.*
2. *Σ is an alphabet representing the* input symbols*, with an additional blank symbol $\sqcup$ not included in Σ. We will write $\Sigma' = \Sigma \cup \{\sqcup\}$, which represents all possible tape symbols.*
3. *$\delta : Q \times \Sigma' \to Q \times \Sigma' \times \{L, S, R\}$ is the* transition function*.*

Each Turing machine defines the following computational process: Given an input $x \in \Sigma^*$ of non-blank symbols, $x = s_1 s_2 \cdots s_n$ for some $n \in \mathbb{N}$, the Turing machine tape is initialized as

$$\cdots \sqcup \sqcup s_1 s_2 \cdots s_n \sqcup \sqcup \cdots$$

The TM's state is initialized with q_0, and the reading head is positioned at s_1. At each step, if the current state is q and the current symbol read by the head is s, the instruction $\delta(q, s) = (q', s', a)$ for $a \in \{L, S, R\}$ is applied. This is interpreted as follows: the TM's state is changed to q', the head erases the tape symbol s and writes s' instead, and subsequently, the head moves either left (L), right (R), or stays in place (S). If the TM ever enters the halting state q_{halt}, the computation immediately terminates. We say that $\mathcal{T}$ halts on x and write $\mathcal{T}(x) \downarrow$. In such a case, if the tape contains exactly one contiguous block of non-blank symbols, we consider it to be the output of the computation. Otherwise, we say the output is an empty sequence ε. If we denote the output sequence by $y \in \Sigma^*$, we can interpret $\mathcal{T}$ as a mapping which sends x to y. However, it can happen that throughout the computation, $\mathcal{T}$ never enters the halting state and keeps on applying the instructions indefinitely: we say that $\mathcal{T}$ diverges on x and write $\mathcal{T}(x) \uparrow$. In such a case, we can interpret the mapping induced by $\mathcal{T}$ to be undefined on the input x. The mapping we have just described is a key notion in computability theory and we summarize it in the definition below.

Definition 1.3 (Partial computable function, Total computable function)**.** *Each Turing machine $\mathcal{T} = (Q, \Sigma, \delta)$ defines a partial function $\varphi_{\mathcal{T}} : \Sigma^* \rightharpoonup \Sigma^*$ which is defined for all x such that $\mathcal{T}(x) \downarrow$ and gives the corresponding output of the Turing machine. We say that a partial function $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ is partial computable, if there exists a Turing machine $\mathcal{T} = (Q, \Sigma, \delta)$ such that $\varphi = \varphi_{\mathcal{T}}$. If the domain of φ is equal to the whole Σ^*, we say that φ is total computable. We will often say that φ* is implemented by *$\mathcal{T}$ or that $\mathcal{T}$ induces or computes φ.*

In his original paper [8], Turing gave a detailed discussion of why this formalism captures well our intuitive understanding of what a program/algorithm/computational process is. Around the same time, Kurt Gödel introduced the concept of recursive functions, and Alonso Church the notion of lambda calculus. Interestingly, soon after their publications, all these notions were proven to be equivalent. It thus became clear that the notions are capturing something profound and lead to the famous Church-Turing thesis:

Thesis 1.4 (Church-Turing Thesis)**.** *Every computational process can be implemented by a suitable Turing machine.*

This is classically used, in reverse, as the definition of what a computational process is.

Exercise 1.5 (The Contrarian Machine)**.** *Define a Turing machine with an alphabet $\Sigma = \{0, 1\}$, which outputs the negation of any binary input string $x \in \{0, 1\}^*$.*

Exercise 1.6 (The Copying Machine)**.** *Define a Turing machine with an alphabet* $\Sigma = \{0, 1, \#\}$*, where* $\#$ *symbolizes a delimiting symbol, which outputs* $x\#x$ *for any binary input string* $x \in \{0,1\}^*$*.*

It will be useful to view partial computable functions as operating on natural numbers. For that, we need to define a bijection between Σ^* and $\mathbb{N}$. There are multiple ways of doing this; below, we present a method based on the lexicographical ordering.

Enumerating Sequences Let $\Sigma = \{s_1, \dots, s_k\}$ for some $k \in \mathbb{N}$. We define an arbitrary ordering of the symbols: $s_1 < s_2 < \dots < s_k$ and describe how this induces a linear order of Σ^*. Let $x, y \in \Sigma^*$, we say that $x \prec y$ if either:

1. the length of x is less than the length of y
2. x and y have the same length, and x is strictly smaller than y in the lexicographical ordering induced by $<$.

Thus, we can enumerate the elements of Σ^* from the smallest as follows:

$$\varepsilon, s_1, s_2, \dots, s_k, s_1 s_1, s_1 s_2 \dots.$$

We define the bijection $\beta : \mathbb{N} \to \Sigma^*$ as mapping $n \in \mathbb{N}$ to the n-th smallest element with respect to the ordering $\prec$; $\beta(0) = \varepsilon, \beta(1) = s_1$, etc. It is easy to verify that the function β can be implemented by a suitable Turing machine $\mathcal{T}$; i.e., $\beta = \varphi_{\mathcal{T}}$. Moreover, this also holds for the inverse $\beta^{-1} : \Sigma^* \to \mathbb{N}$.

Exercise 1.7 (Encoding Tuples of Sequences)**.** *Define a bijection* $\gamma : \mathbb{N} \to \Sigma^* \times \Sigma^*$ *such that both* γ *and its inverse can be implemented by a Turing machine.*

This allows us for each Turing machine $\mathcal{T}$ to interpret the partial function $\varphi_{\mathcal{T}}$ as a function $\varphi_{\mathcal{T}} : \mathbb{N} \rightharpoonup \mathbb{N}$. We note that each computable function (partial or total) can be implemented by multiple different Turing machines. Suppose that $\mathcal{T}$ defines a partial function $\varphi_{\mathcal{T}}$. Then, we can amend $\mathcal{T}$ with extra states or alphabet symbols that will never be reached or used, and add arbitrary instructions using these extra symbols. Thus, modifying $\mathcal{T}$ without changing the computable function it induces.

The terminology related to partial and total computable functions is quite diverse, and typically varies based on the context. An enumeration of a set S is a surjective function $\varphi : \mathbb{N} \to S$. We say such an enumeration is *effective* if φ is a total computable function. A set $S \subseteq \mathbb{N}$ is *decidable* or *recursive* if its characteristic function $\chi_S : \mathbb{N} \to \{0, 1\}$ is a total computable function. A set S is called *recursively enumerable* if there exists a partial computable function f such that for all $s \in \mathcal{S}$ $f(s) = 1$ and $f(s)$ need not be defined or is different from 1 for $s \notin S$. We summarize the notational situation in Table 1.

Object	**Implemented by a TM**	**Implemented by a TM that always halts**
function $f : \mathbb{N} \to \mathbb{N}$	partial computable	total computable
set $S \subseteq \mathbb{N}$	recursively enumerable	recursive / decidable
enumeration $f : \mathbb{N} \to S$	–	effective

Table 1: Overview of notation for "computable objects".

Exercise 1.8 (Different Points of View on Recursively Enumerable Sets). *There are many different equivalent definitions of recursively enumerable sets. We mention three variants: $S \subseteq \mathbb{N}$ is recursively enumerable if:*

1. *There exists a Turing machine that, from an empty input, keeps progressively listing exactly all elements of S on the tape.*
2. *There exists a partial computable function f such that $S = \operatorname{dom}(f)$.*
3. *There exists a partial computable function f such that $S = \operatorname{rng}(f)$.*

Prove these are indeed equivalent to the initial definition we gave.

We can readily generalize the notion of a Turing machine to accommodate for multiple input or output sequences. For instance, one can introduce a special alphabet symbol # on top of the given tape symbols from Σ and consider a tape sequence $x_1\#x_2\#\cdots\#x_n$ to represent the tuple $(x_1, x_2, \ldots, x_n)$ where each $x_i \in \Sigma^*$, $1 \leq i \leq n$. Turing machines with n inputs and m outputs induce partial computable functions $\varphi : (\Sigma^*)^n \rightharpoonup (\Sigma^*)^m$ that can be readily interpreted as functions from $\mathbb{N}^n$ to $\mathbb{N}^m$.

1.2 Gödel Numbering and Universal Turing Machines

In this section, we will discuss a crucial fact: there are only countably many Turing machines and their enumeration can be done effectively: i.e., there exists a Turing machine, which, upon receiving $i \in \mathbb{N}$, outputs the description of the i-th Turing machine. Since we have already described how to enumerate sequences, we are left with showing that each Turing machine can be identified with a sequence of symbols from some fixed alphabet S.

Each triple $\mathcal{T} = (Q, \Sigma, \delta)$ defining a Turing machine consists of finite objects: both Q and Σ are finite sets, moreover, δ is characterized by a finite set of instructions $\{(q, s, q', s', a) \mid q \in Q, s \in \Sigma', (q', s', a) = \delta(q, s)\}$. We show that $\mathcal{T}$ can be described by a finite sequence of symbols from the set $S = \{0, 1, (,), |\}$.

- We can represent each state from Q as a finite sequence of 0s and 1s. For instance, assuming 0 always represents q_0 and 1 represents q_{halt}.
- Similarly, we can represent the tape symbols Σ as binary sequences.
- We can represent L, S and R, for instance by sequences $100, 010$ and 001.
- Thus, every instruction of δ can be expressed as $(\cdot \mid \cdot \mid \cdot \mid \cdot \mid \cdot)$ where each $\cdot$ is an element of $\{0, 1\}^*$. For instance, $(0|11|01|10|001)$ represents an instruction stating that if the current state is q_0 and the current symbol read is (represented by) 11, then the new state should be 01, the symbol to be written is 10 and the reading head should move right.

Then, $\mathcal{T}$ can be encoded as a sequence which can look, for example, as follows

$$(0|10|1|00|010)(0|10|1|00|100)(01|10|1|00|100)\cdots(10011|10|1|00|001).$$

We will denote this sequence as $[\mathcal{T}]$. It is not difficult to see that there exists a Turing machine that, upon receiving a sequence $x \in S^*$, outputs a 1 if it is a valid representation of a Turing machine, and outputs a 0 otherwise. The TM has to simply check the syntactical correctness (checking that every matching pair of parentheses contains a 5-tuple of binary sequences separated by |, etc.), and verify that the instructions yield a deterministic function δ.

Definition 1.9 (Standard description of Turing Machines)**.** *Let $\mathcal{T}$ be a Turing machine and $S = \{0, 1, (,), |\}$. We will call the sequence $[\mathcal{T}] \in S^*$ described above the standard description of $\mathcal{T}$.*

It is entirely natural to regard a Turing machine as a finite string over a fixed alphabet. After all, this mirrors how we treat "real-world programs": a Python program, for instance, is its source code—a string of characters drawn from the language's vocabulary. A program becomes dynamic only when interpreted or executed by some external mechanism (a compiler, a processor, or ultimately the laws of physics). In itself, however, it remains a static, syntactic object. Thinking of Turing machines in this way allows us to speak uniformly about "machines" and "descriptions": both are simply strings, some of which describe processes that act on other strings. We can summarize in the following observation.

Observation 1.10. *Let $\mathbf{TM} \subseteq S^*$ be the set of all Turing machines' standard descriptions. Then, there exists a total computable function $G : \mathbb{N} \to \mathbf{TM}$ which is a bijection, and its inverse is also total computable.*

Proof. We describe a Turing machine $\mathcal{T}$ which implements G as follows: upon receiving n as input, $\mathcal{T}$ proceeds by setting a counter on the tape to 0 and by progressively enumerating S^*. For each such sequence $\bar{s} \in S^*$, the TM checks whether it is a valid description of a Turing machine and if so, increases its tape counter by one. Once the counter reaches n, $\mathcal{T}$ has generated the correct output. Because each Turing machine has a finite description, the n-th description appears after finitely many steps in the enumeration of S^*, so $\mathcal{T}$ indeed halts for every n. Checking that the inverse is also total computable is now not difficult. $\square$

If $G(n) = [\mathcal{T}]$ we will call n the Gödel number of $\mathcal{T}$. Further, we denote by φ_n the partial computable function implemented by the Turing machine with Gödel number n. This gives us an enumeration of the set of partial recursive functions. Note that this enumeration contains each function multiple (in fact infinitely many) times, since the exact same function is implemented by infinitely many Turing machines.

The term Gödel numbering and the function G in the previous observation refer to Kurt Gödel's work on the incompleteness theorems [3] where he describes ways to effectively encode progressively more complicated objects: sequences of finite symbols, formulas, mathematical proofs, etc. into natural numbers.

There are numerous ways to define a Gödel numbering, and we note that above, we merely give an example of one possible way to do this. In the remark below, we briefly mention a variant that behaves well with respect to Turing machine composition: we will use this fact later when discussing quines.

We remark that it is not hard to see that each family consisting of all partial computable functions with n inputs and m outputs can also be enumerated. The following theorem describes the relationship between partial computable functions with a single input and with two inputs, essentially stating that one input of the bivariate function can be effectively converted as a parameter for a univariate function.

Theorem 1.11 (Parameter Theorem)**.** *Fix an enumeration ψ_n of partial computable functions $\psi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Then there exists a total computable function $s : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for each $n \in \mathbb{N}$ and inputs x, y we have that $\psi_n(x, y) = \varphi_{s(n,x)}(y)$. Moreover, s can be chosen to be injective.*

Proof. For now, we focus on constructing s which is total computable and not necessarily injective. We sketch out an informal description of the Turing machine $\mathcal{S}$ with two inputs, which implements $\varphi_{s(n,x)}$. First, $\mathcal{S}$ obtains from n the description of a Turing machine $\mathcal{T}$ implementing $\psi_n(x, y)$.

Then, $\mathcal{S}$ modifies this description to "hardcode" the value of the first input to be exactly the supplied parameter x. Informally, a Python analogue of the function $\mathcal{S}$ is illustrated below.

```python
def s(f, a):
    def new_program(b):
        return f(a, b)
    return new_program
```

$\square$

To show that the function s from the previous theorem can be injective, we will use the following lemma, which tells us that given an index n of a partial computable function, we can efficiently construct an arbitrarily large integer which indexes the same function.

Lemma 1.12 (Padding Lemma). *There exists a total computable function $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which on input (n, a) outputs an integer n' such that $\varphi_n = \varphi_{n'}$ and $n' > a$.*

Proof. The function p can be implemented by a Turing machine which simply recovers the description $G(n) = [\mathcal{T}]$ and pads $[\mathcal{T}]$ this with additional instructions that will never be used, in order to obtain a description of a Turing machine $\mathcal{T}'$ implementing the same computable function as $\mathcal{T}$, but with a Gödel number exceeding b. $\square$

Now we can finish the proof of the Parameter Theorem.

Proof. We have already constructed a total computable function $s : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for each $n \in \mathbb{N}$ and inputs x, y we have that $\psi_n(x, y) = \varphi_{s(n,x)}(y)$. We now construct $s' : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which will be an injective version of s. The function s' uses a fixed enumeration of the input tuples $\gamma : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, with $\gamma(0)$ being the least tuple, $\gamma(1)$ the next one, etc. Now, for the input tuple $\gamma(0) = (n_0, x_0)$ we simply define $s'(n_0, x_0) = s(n_0, x_0) = a_0$. For $\gamma(1) = (n_1, x_1)$ we let s' to first compute a_0. If $s(x_1, y_1) \leq a_0$, we use the Padding Lemma to find a_1 such that $\varphi_{a_1} = \varphi_{s(n_1,x_1)}$ and $a_1 > a_0$. Following this procedure inductively, we ensure that the outputs of s' are strictly increasing with respect to the input enumeration, and thus, that s' is injective. $\square$

Example 1.13. *We define a Python function that sums two numbers, and parametrize the first input to be 5.*

```python
def summed(x, y):
    return x+y

new_program = s(summed,5)
print(new_program(10))
```

Then, the last line outputs 15.

Now we can proceed to a fundamental concept in the theory of computation.

Definition 1.14 (Universal Turing Machine). *We define a partial function $u : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ as $u(i, n) = \varphi_i(n)$ and call it the* universal partial computable function. *It is indeed not difficult to see that there exists a Turing machine $\mathcal{U}$ which computes it: $\mathcal{U}$ upon receiving (i, n) as input computes $G(i) = [\mathcal{T}]$ and uses this description to simulate $\mathcal{T}$ on the input n. Note that if $\mathcal{T}(n) \uparrow$ then $\mathcal{U}(i, n) \uparrow$ as well. We call any Turing machine which implements the universal function u a* universal Turing machine*.*

1.2.1 Historical Window: General Computers

Turing's construction of a universal machine in [8] was groundbreaking because it showed that a single, fixed machine could simulate any other computing procedure when given an appropriate description of that procedure - establishing, for the first time, the principle of stored-program universality. This idea is directly analogous to modern computers, whose hardware is general-purpose and whose behaviour is determined by software rather than by rewiring the machine. The influence of Turing's model on real computing architecture is well documented: John von Neumann explicitly acknowledged Turing's 1936 work when formulating the design principles of the EDVAC (one of the earliest fully electronic, general-purpose computers) and articulating the stored-program architecture that later came to bear his name: the von Neumann architecture.

1.3 The Halting Problem

There are clearly uncountably many functions $f : \mathbb{N} \rightharpoonup \mathbb{N}$, however, there are only countably many Turing machines. This implies the existence of (infinitely many) functions which are not partial computable. In this section, we construct a famous example of such a function: the halting problem.

Theorem 1.15. *Let $h : \mathbb{N} \to \mathbb{N}$ be a function defined as follows:*

$$h(n) = 1 \text{ if } \varphi_n \text{ is defined on } n \tag{1}$$

$$h(n) = 0 \text{ if } \varphi_n \text{ is not defined on } n \tag{2}$$

Then, h is a function defined for every input but it is not total computable.

Proof. For contradiction, we will assume that h is total computable. Thus, that there exists a TM $\mathcal{T}$ which implements h, and since h is defined everywhere, $\mathcal{T}$ halts on every input. We construct a new partial function $\tilde{h} : \mathbb{N} \rightharpoonup \mathbb{N}$ as follows:

$$\tilde{h}(n) = 0 \qquad \text{if } h(n) = 0 \tag{3}$$

$$\tilde{h}(n) \text{ is not defined} \quad \text{if } h(n) = 1. \tag{4}$$

We can define a Turing machine $\tilde{\mathcal{T}}$ which, on input n first computes $h(n)$, and outputs a 0 when $h(n) = 0$, and diverges if $h(n) = 1$ (it can for instance enter a never-ending loop akin to "while True: print(1)"). Then, $\tilde{\mathcal{T}}$ implements $\tilde{h}$, so $\tilde{h}$ is a partial computable function and thus equal to φ_i for some $i \in \mathbb{N}$. Now, let us see what happens for $\varphi_i(i)$:

If $h(i) = 0$ then $\varphi_i(i)$ is not defined by (2); this is impossible since $\varphi_i(i) = \tilde{h}(i) = 0$ by (3).

If $h(i) = 1$ then $\varphi_i(i)$ is defined by (1); this is impossible since $\varphi_i(i) = \tilde{h}(i)$ is not defined by (4).

Thus, we arrived at a contradiction. □

One can associate with h the set $H \subseteq \mathbb{N}$ where $H = \{n \mid h(n) = 1\}$ and reframe Theorem 1.15 as a statement that H is not decidable. We will often encounter the expression that a certain problem is undecidable.

Remark 1.16 (Diagonalization Argument)**.** *We remark that the reasoning in the previous proof is essentially analogous to Cantor's diagonalization argument. Indeed, let us create a table which stores information about all partial computable functions $\varphi : \mathbb{N} \to \mathbb{N}$. The rows enumerate the functions, and the columns index the inputs.*

	0	1	2	$\cdots$	n	$\cdots$
φ_0	$\underline{\varphi_0(0)}$	$\varphi_0(1)$	$\varphi_0(2)$	$\cdots$	$\varphi_0(n)$	$\cdots$
φ_1	$\varphi_1(0)$	$\underline{\varphi_1(1)}$	$\varphi_1(2)$	$\cdots$	$\varphi_1(n$	$\cdots$
φ_2	$\varphi_2(0)$	$\varphi_2(1)$	$\underline{\varphi_2(2)}$	$\cdots$	$\varphi_2(n)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
φ_n	$\varphi_n(0)$	$\varphi_n(1)$	$\varphi_n(2)$	$\cdots$	$\underline{\varphi_n(n)}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\ddots$

Table 2: Enumeration table of partial computable functions.

Table 2 highlights the diagonal entries. The diagonal is itself a partial computable function $d(n) = \varphi_n(n)$. The diagonalization argument typically uses some clever transformation of d to obtain a new function $\tilde{d}$ which differs from d at every entry. As a result, $\tilde{d}$ cannot be represented by any row of the table.

This is exactly the case in the proof of Halting problem's undecidability. The function $\tilde{h}$ satisfies

$$\tilde{h}(n) \neq \varphi_n(n) \quad \textit{for all } n \in \mathbb{N}.$$

Thus, the construction of $\tilde{h}$ is exactly a way to diagonalize out of the set of partial computable functions, as such, $\tilde{h}$ cannot be partial computable. We conclude that if h was computable, then necessarily $\tilde{h}$ would have to be partial computable as well, which leads to a contradiction.

Exercise 1.17. *Using a variant of the diagonalization argument, show that the set $S = \{n \in \mathbb{N} \mid \varphi_n : \mathbb{N} \to \mathbb{N}$ is a total computable function$\}$ is not decidable.*

Having a concrete example of an undecidable problem is very useful, since it can now be used to show that other functions are not total computable. The argument follows a reduction technique: one simply needs to show that if the function at hand was computable, the halting problem would be too; i.e., reducing the given function to the halting problem via a Turing machine. We give an example of this below.

Proposition 1.18. *We say that two Turing machines $\mathcal{T}_1$ and $\mathcal{T}_2$ are equivalent if $\varphi_{\mathcal{T}_1} = \varphi_{\mathcal{T}_2}$ and we write $\mathcal{T}_1 \cong \mathcal{T}_2$. The problem whether any two Turing machines are equivalent is undecidable; i.e., the set $\{([\mathcal{T}_1], [\mathcal{T}_2]) \mid \mathcal{T}_1, \mathcal{T}_2 \in \mathbf{TM}, \mathcal{T}_1 \cong \mathcal{T}_2\}$ is not decidable.*

Proof. Suppose for contradiction that there exists a Turing machine $\mathcal{D}$ such that $\mathcal{D}([\mathcal{T}_1], [\mathcal{T}_2]) = 1$ whenever $\mathcal{T}_1 \cong \mathcal{T}_2$ and $\mathcal{D}([\mathcal{T}_1], [\mathcal{T}_2]) = 0$ otherwise. We define a TM $\mathcal{H}$ on input n as follows:

1. Compute the description $G(n)$ of the n-th Turing machine $\mathcal{T}$.
2. Compute the description of a TM $\mathcal{T}'$ which is defined as follows. On any input $m \neq n$, $\mathcal{T}'$ halts with output 0. On input n, $\mathcal{T}'$ simulates $\mathcal{T}$ and outputs $\mathcal{T}'(n) = 1$ if $\mathcal{T}(n)\downarrow$, otherwise $\mathcal{T}'(n)\uparrow$.

3. Compute the description of a TM δ_n which is defined as follows $\delta_n(n) = 1$ and $\delta_n(m) = 0$ for all $m \neq n$.

4. Simulate $\mathcal{D}([\mathcal{T}'], [\delta_n])$ and output that result.

We see that if $\mathcal{T}(n) \downarrow$ then $\mathcal{T}' \cong \delta_n$ so $\mathcal{D}$ outputs 1. If $\mathcal{T}(n) \uparrow$ then $\mathcal{T}'$ behaves differently from δ_n on input n, so $\mathcal{D}$ outputs 0. Therefore, $\mathcal{H}$ is a well-defined Turing machine that decides the halting problem, yielding the contradiction. □

Exercise 1.19. *Using the reduction technique, show that the set* $\{n \in \mathbb{N} \mid \operatorname{dom}(\varphi_n) = \emptyset\}$ *is not decidable.*

1.4 Quines

We have now developed enough terminology to return to the thought exercise at the beginning of this chapter. Let us state a simpler, yet deeply related, version: Suppose that the machine already has a blank blueprint in its construction arms and our task is to design a program that will instruct the machine to write the exact same program on the new blueprint. We realize that we can reformulate this Question in the language we have just developed:

Can we define a Turing machine $\mathcal{Q}$ that outputs its own description $[\mathcal{Q}]$ from an empty input?

Or, in the modern programming language context: Can we define a program that outputs its own source code from an empty input? We call such a program a *quine* after W.V.O. Quine, a philosopher working on set theory, logic, and self-reference [5].

First, we discuss in detail how quines relate to our simplified blueprint problem. Suppose we define the following partial function:

$$\tilde{u}(n, x) = (n, \varphi_n(x)).$$

It is clear that $\tilde{u}(n, x)$ can be implemented by a Turing machine $\tilde{\mathcal{U}}$: it simply computes the description of the n-th Turing machine (with a single input), and using its instructions, computes $\varphi_n(x)$. The only difference between $\tilde{u}$ and the universal function u from Definition 1.14 is that in this case, if $\tilde{u}$ halts, it does not erase n and instead considers it a part of its output. Now, if $\mathcal{Q}$ is a Turing machine outputting its own description, then $\tilde{U}$ defines a process which on input $([\mathcal{Q}], \varepsilon)$ outputs $([\mathcal{Q}], [\mathcal{Q}])$. Thus, in some sense we can interpret $\tilde{\mathcal{U}}$ as the "machine" and $[\mathcal{Q}]$ as the blueprint content which gets "replicated by $\tilde{\mathcal{U}}$". We note that with respect to a universal Turing machine $\mathcal{U}$, $\mathcal{Q}$ is a (sort of) fixed point since $\mathcal{U}$ defines a process which maps $([\mathcal{Q}], \varepsilon)$ to $[\mathcal{Q}]$.

We encourage the reader to try constructing a quine in their favourite programming language. To make this problem truly challenging, we will seek a general solution that does not exploit any particularity in the programming language's architecture. It is difficult to precisely define this requirement, so below, we show a few canonical examples of quines that exploit a particular property of Python's design.

Example 1.20. *[Exploiting language specific error messages] Running the following code, stored in a file called* `is_it_a_quine.py`*, yields exactly its source code, though it would be far-fetched to claim this program creates its own description: it merely exploits the design of Python's error messages.*

```
File "is_it_a_quine.py", line 1
    File "is_it_a_quine.py", line 1
        ^

SyntaxError: invalid syntax
```

The next two examples rely on a more subtle particularity: the fact that the operating system which ultimately initiates the execution of the code has access to the file system, including the file containing the program's source. This should technically count as additional input. The programs below leverage this fact and avoid a direct construction of their own description.

Example 1.21. *[Loading the program's file] Running the following code, stored in a file called* `is_it_a_quine.py`*, again, yields exactly its source code. However, the code simply refers to its description stored in memory without directly creating it.*

```python
print(open("is_it_a_quine.py").read())
```

Example 1.22. *[Using "introspective" functions that return the program's source code] Running this code also yields the exact same output. However, the construction follows the same principle as Example 1.21.*

```python
def is_it_a_quine():
    import inspect
    print(inspect.getsource(is_it_a_quine))
    print("is_it_a_quine()")

is_it_a_quine()
```

Intuitively, we want to achieve a program that truly recreates its own source code, irrespective of the particularities of the architecture of the machine that executes it. Below, we suggest an informal definition of a universal quine construction.

Definition 1.23. *[Informal Definition of a Universal Quine Construction] A universal quine construction is a Turing machine $\mathcal{P}$ which, upon receiving the description $[\mathcal{U}]$ of an arbitrary universal Turing machine $\mathcal{U}$ halts with the output $[\mathcal{Q}_{\mathcal{U}}]$. Here, $\mathcal{Q}_{\mathcal{U}}$ is a (non-empty) quine with respect to $\mathcal{U}$; i.e., $\mathcal{U}([\mathcal{Q}_{\mathcal{U}}], \varepsilon) = [\mathcal{Q}_{\mathcal{U}}]$.*

One's first attempt at writing such a "universal quine" might look as follows:

```python
line1 = 'line1 = "???"\nprint(line1)'
print(line1)
```

Where we might try to replace ??? with some clever text to finish the construction, a natural candidate is replacing ??? with the current source code. After solving some technical issues with nested quotes (typically by replacing the inner simple quotes by "chr(39)" and double quotes by "chr(34)"), we realize our current program will print the source code of our previous version, not the current one. Thus, unless we come up with some structural change in our code, this will lead to an infinite series of updates. We now present a solution in the context of Turing machines, following the remarkably clear narrative of Sipser [6].

Proposition 1.24 (Quine Existence)**.** *There exists a Turing machine $\mathcal{Q}$, which outputs its own description $[\mathcal{Q}]$ from an empty input.*

Proof. We will "split $\mathcal{Q}$" into two Turing machines $\mathcal{A}$ and $\mathcal{B}$ which reproduce each other's code while avoiding a cyclic definition. Subsequently, we define $\mathcal{Q} = \mathcal{A}\mathcal{B}$ to be a (canonical choice of a) Turing machine which implements the function $\varphi_{\mathcal{B}} \circ \varphi_{\mathcal{A}}$ and show that $\mathcal{Q}$ outputs $[\mathcal{Q}]$ from an empty input. Intuitively, $\mathcal{A}$ will have the role of data, and $\mathcal{B}$ will have a functional role that acts on the data.

First, we denote by $\mathcal{P}_c$ the Turing machine implementing a process that, no matter what its input is, halts with output c. Next, we define a Turing machine $\mathcal{B}$ which upon receiving an input x computes $[\mathcal{P}_x]$. If $x = [\mathcal{T}]$ is a valid description of a Turing machine, $\mathcal{B}$ outputs the description $[\mathcal{P}_x\mathcal{T}]$ of a (canonical choice of a) Turing machine implementing the function $\varphi_{\mathcal{T}} \circ \varphi_{\mathcal{P}_x}$. If x was not a valid Turing machine description, $\mathcal{B}$ halts with empty output. We leave it as an exercise for the reader to verify that $\mathcal{B}$ can indeed be implemented as a Turing machine that halts on every input. Below, we illustrate the intuition behind the definition of $\mathcal{A}$ and $\mathcal{B}$ in Python (we resort to the lambda notation so we do not have to deal with newline symbols).

```python
A = lambda: "P = lambda m: return m"
#A is a constant function which outputs the string P = lambda m: return m"
#in this case, this string happens to be a valid Python program

B = lambda m: "A = lambda: "+chr(34)+m+chr(34)+"\n"+m+"\n"+"print(P(A()))"
#b is a function which outputs a string consisting of 3 parts:
#first, it defines the constant function A through
#"A = lambda: "+chr(34)+m+chr(34) (chr(34) represents the double quotes)
#second, it outputs m itself (this is the program P)
#third, assuming m is a description of a valid program named P
#it outputs the composition P(A())
print(B(A()))
```

Lastly, we define $\mathcal{A}$ to be exactly $\mathcal{P}_{[\mathcal{B}]}$. We have avoided the cyclic dependency since $\mathcal{B}$ does not rely on $\mathcal{A}$. The Turing machine $\mathcal{Q} = \mathcal{A}\mathcal{B}$ operates as follows on an empty input: first $\mathcal{A}$ outputs $[\mathcal{B}]$, then $\mathcal{B}$ reads $[\mathcal{B}]$, computes $[\mathcal{P}_{[\mathcal{B}]}] = [\mathcal{A}]$, and outputs $[\mathcal{P}_{[\mathcal{B}]}\mathcal{B}] = [\mathcal{A}\mathcal{B}]$. Thus, the output of $\mathcal{Q}$ is exactly $[\mathcal{A}\mathcal{B}] = [\mathcal{Q}]$. □

The key result of Proposition 1.24 is a "universal recipe" to construct quines no matter what is the particular architecture of $\mathcal{U}$: it will work in any rich enough programming language even without access to the file system or any language specific messages.

Exercise 1.25 (Universal Quine Construction)**.** *Use the construction outlined in Proposition 1.24 to write a quine in your favourite programming language.*

1.5 Kleene's Recursion Theorems

Ultimately, we would like to construct a program $\mathcal{R}$ that will not only instruct the machine to write an exact copy of $\mathcal{R}$ on a new blank blueprint, but that will also specify the machine's function in the process of Mars terraformation. Thus, the quine must also include a program determining (an arbitrary) computable function. The fact that such a construction is always possible is precisely what the Kleene Recursion Theorems guarantee. These are two fixed-point theorems proved by Stephen Kleene [4], a student of Alonso Church. Although the two theorems are logically equivalent (a fact we will establish later), we state them separately because each highlights a different context of self-reference: the first emphasizes fixed points of total computable functions, while the second emphasizes a program's ability to obtain and use its own description as data.

Theorem 1.26 (Kleene's First Recursion Theorem)**.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a total computable function. Then, there exists an $n \in \mathbb{N}$ such that $\varphi_n = \varphi_{f(n)}$.*

Theorem 1.27 (Kleene's Second Recursion Theorem)**.** *Let $\psi : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ be a partial computable function. Then, there exists an $n \in \mathbb{N}$ such that for all inputs $x \in \mathbb{N}$ it holds that $\varphi_n(x) = \psi_{\mathcal{T}}(x, n)$.*

We first show a few applications of the Kleene's theorems. Then, we prove the Second Recursion Theorem, and subsequently, we prove the First Recursion Theorem is equivalent to it. Finally, at the end of this section, we show a direct proof of the First Recursion Theorem which may seem rather unintuitive at a first glance. First, let us reformulate Kleene's Second Recursion Theorem in the language of Turing machines.

Theorem 1.28 (Kleene's Second Recursion Theorem')**.** *Let $\psi : S^* \times S^* \rightharpoonup S^*$ be a partial computable function. Then, there exists a Turing machine $\mathcal{R}$ with a single input and output such that for all $x \in S^*$ it holds that $\varphi_{\mathcal{R}}(x) = \psi(x, [\mathcal{R}])$.*

Existence of Quines It is easy to see that Proposition 1.24 is a direct consequence of the Second Recursion Theorem: For $\psi(x, y) = y$ it is clear that $\varphi_{\mathcal{R}}(x) = [\mathcal{R}]$ for any input x. Thus, $\mathcal{R}$ is exactly a Turing machine outputting its own description. We encourage the readers to think how the First Recursion Theorem implies the quine existence.

Exercise 1.29 (Quine as a fixed point)**.** *How can the Kleene's First Recursion Theorem be used to prove the existence of a Turing machine $\mathcal{R}$ that upon receiving any input, halts and outputs $[\mathcal{R}]$?*

Existence of Quines with Functionality Let $n \in \mathbb{N}$ and define $\psi(x, y)$ to output the pair encoded as $\varphi_n(x)\#y$ if $\varphi_n(x)$ is defined, otherwise $\psi(x, y)$ is undefined as well. Then, the Second Recursion Theorem guarantees the existence of a TM $\mathcal{R}$ which is exactly an "enriched quine" both computing φ_n and outputting its own description.

We will now sketch the proof of the Second Recursion Theorem.

Proof of Kleene's Second Recursion Theorem. Suppose $\psi : S^* \times S^* \rightharpoonup S^*$ is the given partial computable function implemented by some Turing machine $\mathcal{T}$. We will construct a Turing machine $\mathcal{R}$ that on input x outputs the same output as $\mathcal{T}$ does on input $(x, [\mathcal{R}])$.

The outline is analogous to the one of Proposition 1.24, except that now, we will define $\mathcal{R}$ as a composition of three Turing machines: $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{T}$.

For each $y \in S^*$ let $\mathcal{P}'_y$ denote the Turing machine that, upon receiving any input x, returns the string $x\#y$. This time, we define $\mathcal{A} = \mathcal{P}'_{[\mathcal{B}\mathcal{T}]}$. Next, we define $\mathcal{B}$ on an input pair $x\#y$ to compute the description $[\mathcal{P}'_y]$, and check whether y is the valid description of some Turing machine $\mathcal{T}'$. If not, it returns empty output. If yes, it computes the description of the Turing machine $[\mathcal{P}'_y\mathcal{T}']$ and return the string $x\#[\mathcal{P}'_y\mathcal{T}']$. Now, let $\mathcal{R} = \mathcal{A}\mathcal{B}\mathcal{T}$. Then, $\mathcal{R}$ acts on the initial tape containing input x as follows:

1. Stage 1: Executing $\mathcal{A}$ on input x yields $x\#[\mathcal{B}\mathcal{T}]$.
2. Stage 2: Executing $\mathcal{B}$ on input $x\#[\mathcal{B}\mathcal{T}]$, yields $x\#[\mathcal{P}'_{[\mathcal{B}\mathcal{T}]}\mathcal{B}\mathcal{T}] = x\#[\mathcal{A}\mathcal{B}\mathcal{T}] = x\#[\mathcal{R}]$.
3. Stage 3: Executing $\mathcal{T}$ on input $x\#[\mathcal{R}]$.

Therefore, it is clear that $\varphi_{\mathcal{R}}(x) = \psi_{\mathcal{T}}(x, [\mathcal{R}])$. □

Exercise 1.30 (Universal Quine with Functionality Construction)**.** *Use the construction outlined in the proof of the Kleene's Second Recursion Theorem to write a quine in your favourite programming language that prints the string "Hello World", on top of printing its own description.*

We now show the two theorems are equivalent.

Proposition 1.31. *The First and Second Kleene's Recursion Theorems are equivalent.*

Proof. First $\implies$ Second: Let $\psi : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ be a partial computable function. Due to the Parameter Theorem, we have the existence of a total computable function $f : \mathbb{N} \to \mathbb{N}$ such that $\varphi_{f(y)}(x) = \psi(x, y)$ for all inputs x and y. Let n be the fixed point of f (in the sense of Kleene's First Recursion Theorem). Then, $\varphi_n(x) = \varphi_{f(n)}(x) = \psi_n(x, n)$.

Second $\implies$ First: We define a Turing machine $\mathcal{T}$ with two inputs and a single output as follows on input pair $(x, y) \in S^* \times S^*$:

1. If y is the valid description of some Turing machine $\mathcal{A}$; i.e., $y = [\mathcal{A}]$, compute its Gödel number $m = G^{-1}([\mathcal{A}])$. Otherwise, halt.
2. Compute $f(m)$.
3. Generate the description $G(f(m)) = [\mathcal{B}]$ of a Turing machine $\mathcal{B}$ with Gödel number $f(m)$.
4. Simulate $\mathcal{B}$ on input x and output that.

Due to the Second Recursion Theorem, we are guaranteed the existence of a TM $\mathcal{R}$ such that $\varphi_{\mathcal{R}}(x) = \psi_{\mathcal{T}}(x, [\mathcal{R}])$ for all inputs x. Essentially, $\mathcal{R}$ obtains its own Gödel number n, computes $f(n)$ and simulates the Turing machine with Gödel number $f(n)$ on input x. Let n be the Gödel number of $\mathcal{R}$. It is straightforward to verify that indeed, n is the fixed point of f: $\varphi_n(x) = \varphi_{\mathcal{R}}(x) = \psi_{\mathcal{T}}(x, [\mathcal{R}]) = \varphi_{f(n)}(x)$. □

Finally, we present the direct proof of Kleene's First Recursion Theorem. As it is rather difficult to develop a good intuition for it, we will follow an excellent exposition to this topic from the textbook of Soare [7, p. 29-30], which will help us interpret the proof as a "failed diagonalization argument".

Proof of Kleene's First Recursion Theorem. We have a fixed total computable function f, for which we are trying to find a "fixed point" $n \in \mathbb{N}$ in the sense that $\varphi_n = \varphi_{f(n)}$. We define a diagonal function $d : \mathbb{N} \to \mathbb{N}$ in the following way:

$$\varphi_{d(u)}(z) = \begin{cases} \varphi_{\varphi_u(u)}(z) & \text{if } \varphi_u(u) \text{ converges,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

By the Parameter Theorem, d is an injective, total computable function. Therefore, $f \circ d$ is also a total computable function, and hence, the exists $v \in \mathbb{N}$ such that $\varphi_v = f \circ d$. We claim that $n = d(v)$ is a "fixed point" for f. Since φ_v is total, $\varphi_v(v)$ is defined and $\varphi_{d(v)} = \varphi_{\varphi_v(v)}$. Now,

$$\varphi_n = \varphi_{d(v)} = \varphi_{\varphi_v(v)} = \varphi_{f(d(v))} = \varphi_{f(n)}.$$

□

Kleene's First Recursion Theorem as a Failed Diagonalization Argument From a modern perspective, Kleene's fixed point theorem can be understood as a "failed attempt" to diagonalize out of the class of partial computable functions. Specifically, we will show that if we consider a suitable table, where each row represents a partial computable function (now with two inputs), we can view the total computable function f as a transformation of the diagonal. Since the class of partial computable functions is robust, the result of the transformation is again represented as another row in the table, resulting in the fixed point of f. We illustrate this in detail below.

Consider an enumeration table, with both rows and columns indexed by $\mathbb{N}$, such that the m-th row corresponds to the following sequence of partial computable functions with one variable:

$$\varphi_{\varphi_m(0)}, \varphi_{\varphi_m(1)}, \varphi_{\varphi_m(2)}, \dots .$$

If $\varphi_m(x)$ is undefined, we define the function $\varphi_{\varphi_m(x)}(y)$ to be undefined for every input y. Though it might be a priori unclear what this table represents, we will show it is essentially just an enumeration of all partial computable functions with two inputs.

Let us fix $m \in \mathbb{N}$ and let us define $\psi(x, y) = \varphi_{\varphi_m(x)}(y)$. Clearly, ψ is a partial computable function and the whole m-th row in our enumeration table corresponds to the sequence of functions $\psi(0, y), \psi(1, y), \psi(2, y), \dots$, which together completely characterize ψ. On the other hand, every partial computable function $\psi(x, y)$ is represented by one of the rows. Indeed, using the Parameter Theorem, there exists a total computable function $s : \mathbb{N} \to \mathbb{N}$ such that $\psi(x, y) = \varphi_{s(x)}(y)$. Since s is a computable function, there exists some $m \in \mathbb{N}$ such that $s = \varphi_m$. Therefore, $\psi(x, y) = \varphi_{\varphi_m(x)}(y)$.

Now that we have a good idea of what the table represents, we will explain the failed attempt to diagonalize out of this table. First, we notice that the total computable function $d : \mathbb{N} \to \mathbb{N}$ defined above as

$$\varphi_{d(u)}(z) = \begin{cases} \varphi_{\varphi_u(u)}(z) & \text{if } \varphi_u(u) \text{ converges,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

is exactly a diagonal in the table, as illustrated below. Table 3 illustrates multiple key features.

	$x = 0$	$x = 1$	$x = 2$	$\cdots$	$x = v$
$\varphi_{\varphi_0(x)}$	$\underline{\varphi_{\varphi_0(0)}}$	$\varphi_{\varphi_0(1)}$	$\varphi_{\varphi_0(2)}$	$\cdots$	$\varphi_{\varphi_0(v)}$
$\varphi_{\varphi_1(x)}$	$\varphi_{\varphi_1(0)}$	$\underline{\varphi_{\varphi_1(1)}}$	$\varphi_{\varphi_1(2)}$	$\cdots$	$\varphi_{\varphi_1(v)}$
$\varphi_{\varphi_2(x)}$	$\varphi_{\varphi_2(0)}$	$\varphi_{\varphi_2(1)}$	$\underline{\varphi_{\varphi_2(2)}}$	$\cdots$	$\varphi_{\varphi_2(v)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\varphi_{f\circ d(x)}$	$\varphi_{f\circ d(0)}$	$\varphi_{f\circ d(1)}$	$\varphi_{f\circ d(2)}$	$\cdots$	$\underline{\varphi_{f\circ d(v)}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
$\varphi_{d(x)}$	$\varphi_{d(0)}$	$\varphi_{d(1)}$	$\varphi_{d(2)}$	$\cdots$	$\varphi_{d(v)}$

Table 3: Enumeration table of partial computable functions with two inputs.

Firstly, the diagonal elements are underlined. Since the diagonal function d is itself total computable, it determines a row of the table indexed by $\varphi_{d(x)}$. The function f acts on this row, and transforms it into the row with index $\varphi_{f\circ d(x)}$. Here, we should interpret f as an arbitrary total computable transformation of the diagonal as an attempt to diagonalize out of the class of partial computable functions present in the table. The attempt is failed since the transformed function is again a part of the table. Since $f \circ d = \varphi_v$ (this is how we defined v), we have that the function $\varphi_{f\circ d(v)}$ lies on the diagonal, concretely that it is the v-th element of the diagonal. Similarly, since d is the diagonal function itself, $\varphi_{d(v)}$ denotes the v-th element of the diagonal. Therefore, $\varphi_{f\circ d(v)} = \varphi_{d(v)}$, and $d(v)$ is the desired "fixed point" of f.

1.6 More Quine Examples

We conclude this section with a final example of quines, inspired by a recent paper [1] that studies self-replication phenomena in a variant of the minimalist programming language *BF*, originally introduced by Urban Müller in 1993. For our purposes, we describe a streamlined version of the computational model used in that work.

Each tape has a finite length K, for some fixed $K \in \mathbb{N}$. At initialization, two such tapes are concatenated, creating a string of $2K$ symbols. There are three pointers placed at the first tape symbol: the instruction pointer, the read head, and the write head. We denote their positions by p, r, and w respectively. At initialization, we have $p = r = w = 0$. We assume the tape is circular and compute the indices of each pointer modulo $2K$. The tape contains the blank symbol (⊔), the symbol `0` or symbols encoding the following instructions:

`<`	$r = r - 1$
`>`	$r = r + 1$
`{`	$w = w - 1$
`}`	$w = w + 1$
`-`	$\text{tape}[r] = \text{tape}[r] - 1$
`+`	$\text{tape}[r] = \text{tape}[r] + 1$
`.`	$\text{tape}[w] = \text{tape}[r]$
`,`	$\text{tape}[r] = \text{tape}[w]$
`[`	if $\text{tape}[r] =$ `0`: i_p jumps to matching `]`
`]`	if $\text{tape}[r] \neq$ `0`: i_p jumps to matching `[`

At each time-step, the instruction at the location p is executed, and subsequently p is increased by 1. The only exception is when the instruction pointer performs a "jump" through the instructions `]` and `[`, in such a case, p is not further increased. If there is not a matching parenthesis, the process halts. Otherwise, the process halts artificially after having performed $T_{\max}$ instructions. Once the program halts, the tape is again split into two sequences of K symbols.

For our purposes, we define a very simplistic quine to be a sequence of K symbols $\bar{q}$, such that, when concatenated with a tape containing K symbols `0`, the program halts, and the resulting tapes are both equal to $\bar{q}$.

Exercise 1.32. *Suppose the initial BF tape consists of the sequence* `{.>` *followed by 7 blank symbols. Write down three time-steps of the computational process.*

A BF Quine. One of the minimal quines identified in this modified BF setup is presented below. We set its length to $K = 18$.

$\bar{q} =$ `[[{.>]-]⊔⊔]-]>.{[[`

In Figure 1 we show the time evolution of the computational process seeded with $\bar{q}$, concatenated with a tape full of zeros.

From Figure 1, it is apparent that the functional core of the quine is represented by the sequence `[{.>]` which keeps on looping and in each loop, it instructs the write head to decrement its position by one, write the current read head symbol, and finally it instructs the read head to increment its position by one. The rest of the quine is arranged in a palindromic way for the construction to give a perfect copy.

Exercise 1.33 (The nature of BF quine)**.** *Do you think the above quine construction is a universal in the sense of Definition 1.23?*

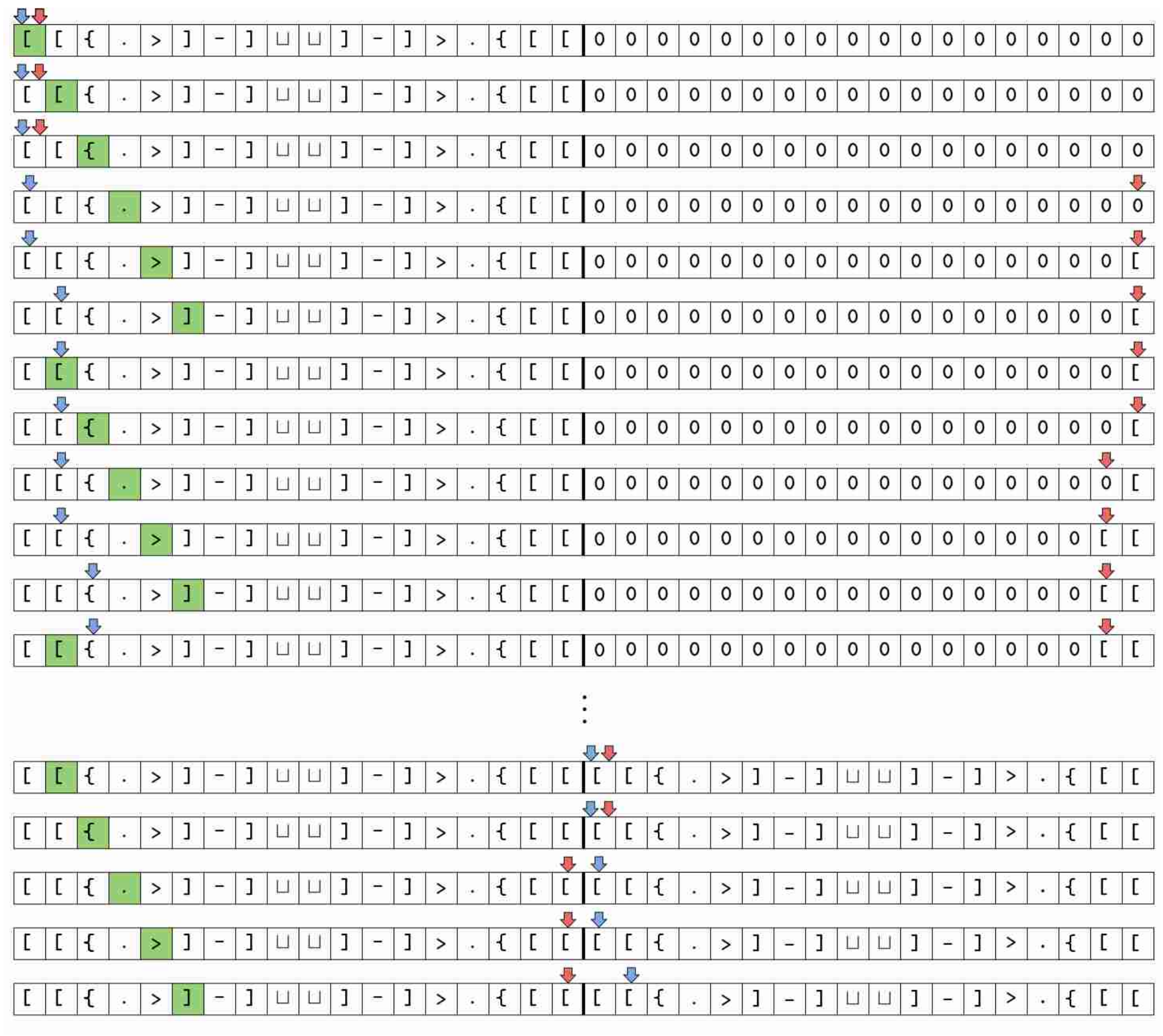

Figure 1: Evolution of the BF computation seeded with the quine $\bar{q}$ concatenated with a tape containing only zeros. Time flows downward, each row is a snapshot of the tape. The green colour shows the instruction pointer's position, the blue arrow represents the read head, the red arrow the write head. After a certain point, the read and write heads just keep symmetrically circulating around the tape, leaving the symbols intact, until the computation is artificially stopped. The outcome consists of two tapes of length 18, both identical to $\bar{q}$.

1.7 Chapter Summary

In this chapter, we have seen a solution to a substantial part of our self-replicating machine construction. Namely, we now know what instructions to put on the machine's blueprint that would specify the machine's function, and would instruct the machine to copy the exact instructions on an empty blueprint.

We have, however, entirely neglected the "physical implementation" of the replication process: how do we construct the machine and the blueprint in some "physically feasible system" so that the machine could execute the blueprint and build not only a perfect copy of the blueprint's content, but also of itself?

In the next chapter, we will see a solution due to John von Neumann in a very idealized model of the physical world based on local parallel interactions. Even though von Neumann draws inspiration from Turing's construction of a universal machine, he does not rely directly on Kleene's results on the self-referential programs, and it is unclear whether he was even aware of them at the time. For this reason, it is best to view von Neumann's results as logically independent — though deeply parallel in spirit — to Kleene's recursion theorems. We will discuss this connection in depth at the end of the next chapter.

References

[1] Alakuijala, J., Evans, J., Laurie, B., Mordvintsev, A., Niklasson, E., Randazzo, E., Versari, L., et al. Computational life: How well-formed, self-replicating programs emerge from simple interaction. *arXiv preprint arXiv:2406.19108* (2024).

[2] Freitas Jr, R. A. A self-reproducing interstellar probe. *Journal of the British Interplanetary Society 33*, 7 (1980), 251–264.

[3] Gödel, K. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme i. *Monatshefte für Mathematik und Physik 38*, 1 (1931), 173–198.

[4] Kleene, S. C. Introduction to Metamathematics.

[5] Quine, W. V. O., et al. The Ways of Paradox. *The ways of paradox and other essays* (1966), 3–20.

[6] Sipser, M. Introduction to the Theory of Computation. *ACM Sigact News 27*, 1 (1996), 27–29.

[7] Soare, R. I. *Turing Computability: Theory and Applications.* Theory and Applications of Computability. Springer-Verlag, Berlin Heidelberg, 2016.

[8] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math 58*, 345-363 (1936), 5.

[9] von Neumann, J. *Theory of self-reproducing automata.* University of Illinois press Urbana, 1966, Editor: A.W. Burks.