Computability Theory: A Very Short Introduction

Amir Tabatabai
Faculty of Humanities, Utrecht University

September 2019

1 Deterministic Finite Automata

As we have explained in the Introduction, computation is a mechanical manipulation of symbols via the local operations of reading, writing and erasing, all governed by a finite set of given rules. In this section we limit ourselves to a very restricted notion of computation in which we are only allowed to read the input without writing and erasing any symbol anywhere. One may hesitantly ask that even if some kind of computation is possible in such a limited situation, how the machine is supposed to report the result of the computation without the minimum power of writing. Because of this very reason, in this section, the computation is limited only to the decision procedures through which the machine computes the truth value of the input statement. The mechanism is as follows: The machine has finitely many internal states that change throughout the computation process. Some of these states are called the accepting states. The machine reads the input, letter by letter, and in each step modifies its current state according to what it just has read and what its rules dictate. The whole process ends when the machine reads the last letter of the input and it accepts the input if it lands in a final state.

Definition 1.1. Let Σ be a finite set. By Σ^* we mean the set of all finite strings (including the zero-length string ϵ) consisting of the elements of Σ.

Definition 1.2. By a language over the alphabet Σ we mean any subset of the set Σ^*.

For instance, the English language is a language over the set $\{a, b, c, \ldots, z\}$ while natural numbers can be considered as a language on the binary set $\{0, 1\}$ via the binary expansion or the unary set $\{1\}$ via identifying n by 1^n.

The other examples include the set $\{a^n b^n | n \geq 0\}$ over the alphabets $\{a, b\}$ or the set $\{1^p | p \text{ is a prime number}\}$ over the alphabet $\{1\}$,.

Definition 1.3. A deterministic finite automaton, DFA, M on the alphabet $\Sigma = \{s_1, \ldots, s_n\}$ with states $Q = \{q_1, \ldots, q_m\}$ is given by a function δ which maps each pair (q_i, s_j), $1 \leq i \leq m$, $1 \leq j \leq n$, into a state q_k, together with a set $F \subseteq Q$. One of the states, usually q_1, is singled out and called the initial state. The states belonging to the set F are called the final or accepting states. δ is called the transition function.

It is usually useful to represent a DFA by a directed graph with the nodes representing the states and edges with the alphabet labels to represent the transition function. If the transition function sends p to q reading the letter s, there will be an edge from p to q with label s. The start state is shown by a node with a small unlabelled in-edge and the accepting states are indicated by double circles.

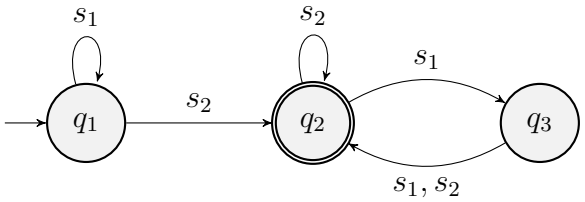

Figure 1: A DFA with the states $\{q_1, q_2, q_3\}$, alphabet $\{s_1, s_2\}$ and $F = \{q_2\}$.

Definition 1.4. Let $q \in Q$ be a state and $w \in \Sigma^*$ be a finite string. By $\delta^*(q, w)$ we mean the state resulting from iterating δ starting from q and reading the letters of w from left to right till it ends. Then M accepts a word w if $\delta^*(q_1, w) \in F$. It rejects w otherwise. Finally, the language accepted (recognized) by M, written $L(M)$, is the set of all $w \in \Sigma^*$ accepted by M, i.e.,:

$$L(M) = \{w \in \Sigma^* | \delta^*(q_1, w) \in F\}.$$

A language is called regular if there exists a deterministic finite automaton which accepts it.

Example 1.5. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{p\}$. Define δ as p for the input (p, a) and q otherwise. Then $L(M) = \{a^n | n \geq 0\}$. See the Figure 2. The reasoning is simple. If M reads a^n, for some n, (including $n = 0$), it goes through the loop over p, n many times. Therefore, it ends up in p and hence M accepts it. But if it includes at least one b, the machine must cross the edge between p and q at

some point and after that it will remain in q till the end. Hence, M can not accept these strings.

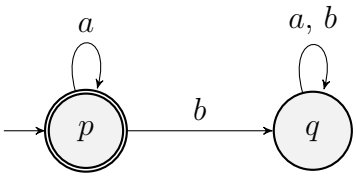

Figure 2: Example 1.5

1.1 Some Examples

Example 1.6. Consider the DFA M on the alphabet $\{a, b\}$ with the state $\{p\}$ such that $F = \emptyset$. Define δ as p for the both inputs (p, a) and (p, b). Then $L(M) = \emptyset$. See the Figure 3.

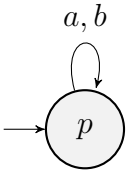

Figure 3: Example 1.6

Example 1.7. Consider the DFA M on the alphabet $\{a, b\}$ with the state $\{p\}$ such that $F = \{p\}$. Define δ as p for the both inputs (p, a) and (p, b). Then $L(M) = \{a, b\}^*$. See the Figure 4.

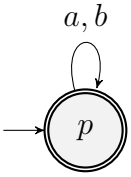

Figure 4: Example 1.7

Example 1.8. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{q\}$. Define δ as q for the input (p, a) and r otherwise. Then $L(M) = \{a\}$. See Figure 5.

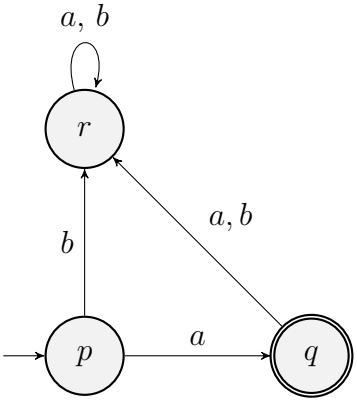

Figure 5: Example 1.8

Example 1.9. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, s\}$ such that $F = \{r\}$. Define δ as q for the input (p, a) and r for the input (q, b) and s otherwise. Then $L(M) = \{ab\}$. See Figure 21.

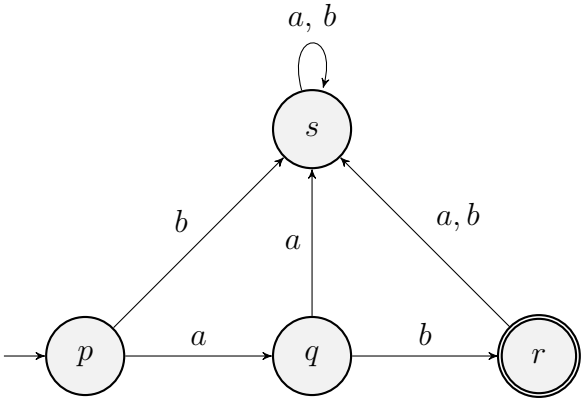

Figure 6: Example 4.5

Exercise 1.10. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{u\}$.

Example 1.11. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, s\}$ such that $F = \{q, r\}$. Define δ as q for (p, a); r for (p, b) and s otherwise. Then $L(M) = \{a, b\}$. See the Figure 7.

Example 1.12. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{r\}$. Define δ as q for the input (p, b), r for the input (p, a), q for both (q, a) and (q, b) and finally the same for r, meaning r for both (r, a) and (r, b). Then $L(M) = \{aw | w \in \{a, b\}^*\}$. See the figure 22.

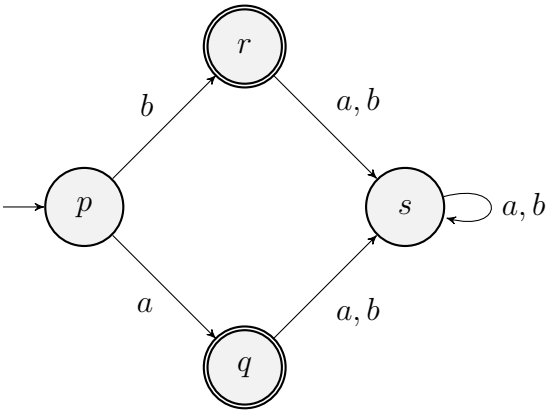

Figure 7: Example 1.11

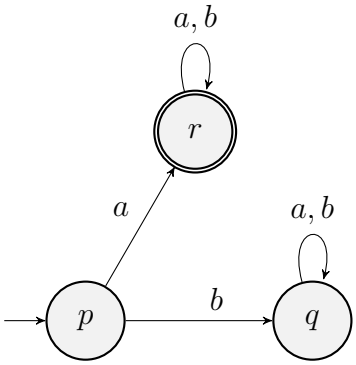

Figure 8: Example 4.9

Exercise 1.13. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, s\}$ such that $F = \{r\}$. Define δ as q for the input (p, a); r for the input (q, b); r for both (r, a) and (r, b) and s for the rest. Then show that $L(M) = \{abw | w \in \{a, b\}^*\}$.

Exercise 1.14. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{uw | w \in \{a, b\}^*\}$.

Example 1.15. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as q for the input (p, a) and (q, a) and p otherwise. Then $L(M) = \{wa | w \in \{a, b\}^*\}$. See the Figure 9.

Exercise 1.16. Define a DFA M such that $L(M) = \{wba | w \in \{a, b\}^*\}$.

Exercise 1.17. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{wu | w \in \{a, b\}^*\}$.

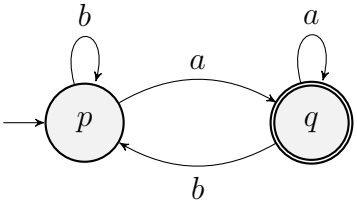

Figure 9: Example 4.11

Example 1.18. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as p for the input (p, a) and q otherwise. Then $L(M) = \{w | w \text{ has at least one } b\}$. See the Figure 10.

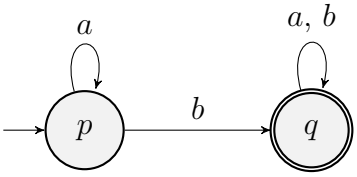

Figure 10: Example 1.18

Exercise 1.19. Compare the Examples 1.5 and 1.18 and answer the following question: Let L be a regular language over the alphabet Σ. Is $\Sigma^* - L$ also regular?

Exercise 1.20. Describe a DFA M that recognizes the strings over $\{a, b\}$ with at least two a's. What about the condition "at most one a"?

Example 1.21. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r, s\}$ such that $F = \{r\}$. Define δ as q for the input (p, a); r for (q, b) and (r, b); q for (q, a) and s otherwise. Then $L(M) = \{a^n b^m | m, n \geq 1\}$. See the Figure 11.

Example 1.22. Consider the DFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as p for the input (q, b); q for (p, a) and r otherwise. Then $L(M) = \{(ab)^n | n \geq 0\}$. See the Figure 12.

Exercise 1.23. Let $u \in \{a, b\}^*$. Define a DFA M such that $L(M) = \{u^n | n \geq 0\}$.

Example 1.24. Consider the DFA M on the alphabet $\{1\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as q for the input $(p, 1)$ and r for $(q, 1)$ and p for $(r, 1)$. Then $L(M) = \{1^{3n} | n \geq 0\}$. Interpreting $\{1\}^*$ as $\mathbb{N}$, the machine recognizes the multiples of three. See the Figure 13.

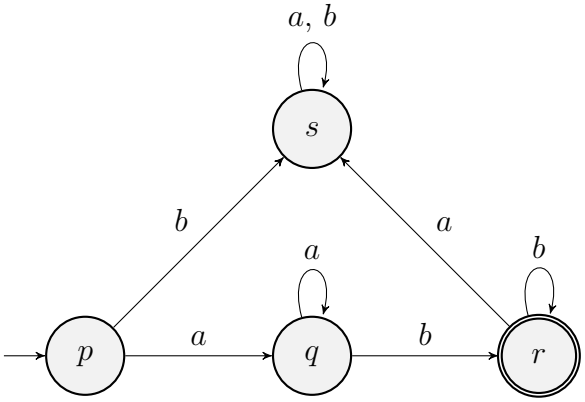

Figure 11: Example 1.21

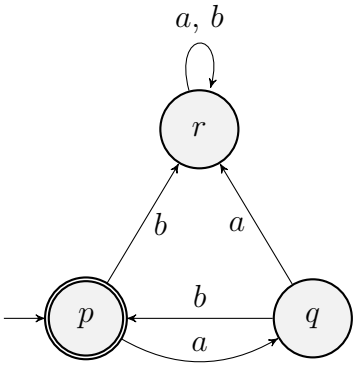

Figure 12: Example 1.22

Exercise 1.25. Define a DFA M on the alphabet $\{1\}$ such that $L(M) = \{1^{3n+1} | n \geq 0\}$.

Exercise 1.26. Let $m \geq 1$ be a natural number and $r < m$. Define a DFA M on the alphabet $\{1\}$ such that $L(M) = \{1^{mn+r} | n \geq 0\}$.

2 Non-deterministic Automata and Regular Operations

In this section we will introduce a generalization of a DFA called a non-deterministic finite automaton or NFA, for short. These models may seem more powerful than the DFA's but they are actually equivalent to them and accept the same languages as the DFA's do. Their advantage, though, lies in the fact that the NFA's are easier and more well-behaved to use. For

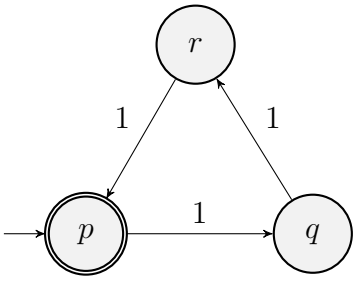

Figure 13: Example 1.24

instance, the class of the new machines is closed under some simple gluing operations that help us to construct more regular languages from the old. We can simulate these operations over the DFA's, as well, but then they are not as transparent as they were for NFA's.

Definition 2.1. A non-deterministic finite automaton, NFA, M on the alphabet $\Sigma = \{s_1, \ldots, s_n\}$ with states $Q = \{q_1, \ldots, q_m\}$ is given by a transition relation $\delta \subseteq Q \times \Sigma \times Q$, together with a set of accepting states $F \subseteq Q$ and the initial state q_1.

Remark 2.2. Note that in a run of an NFA, after reading a letter from the input, the machine may have many possible next states (including zero possibilities), leading to many possible futures. This is why it is called non-deterministic.

Definition 2.3. Let $q \in Q$ be a state and $w \in \Sigma^*$ be a string. By $\delta^*(q, w)$ we mean the *set of all states* resulting from any iteration of δ starting from q and reading the letters of w from left to right till they end. Then M accepts a word w if $\delta^*(q_1, w) \cap F \neq \emptyset$. In other words, it accepts w if there *exists a path of states* following the relation δ and the letters of w from q_1 to F. Finally, the language accepted by M, written $L(M)$, is:

$$L(M) = \{w \in \Sigma^* | \delta^*(q_1, w) \cap F \neq \emptyset\}.$$

2.1 Some Examples

Example 2.4. Consider the NFA M on the alphabet $\{a, b\}$ with the states $\{p, q, r\}$ such that $F = \{r\}$. Define δ as the following relation: $\{(p, a, q), (q, b, r)\}$. Then $L(M) = \{ab\}$. See the Figure 14.

Exercise 2.5. Let $u \in \Sigma^*$. Define an NFA M such that $L(M) = \{u\}$.

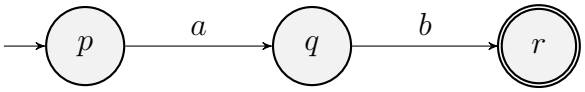

Figure 14: Example 2.4

Example 2.6. Consider the NFA M on the alphabet $\{a, b, c\}$ with the states $\{p, q, r, s\}$ such that $F = \{s\}$. Define δ as the following relation:

$$\{(p, a, q), (p, a, r), (q, b, s), (r, c, s)\}.$$

Then $L(M) = \{ab, ac\}$. See the Figure 15.

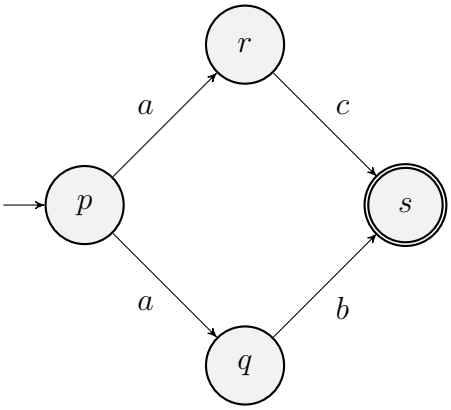

Figure 15: Example 2.6

Exercise 2.7. Let $u_1, \ldots, u_k \in \Sigma^*$. Define an NFA M such that $L(M) = \{u_1, \ldots, u_k\}$.

Exercise 2.8. Let M and N be two NFA's that recognize the languages L_1 and L_2, respectively. Is there an NFA to recognize the language $L_1 \cup L_2$?

Example 2.9. Consider the DFA M on the alphabet $\{a\}$ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as the following relation: $\{(p, a, q), (q, a, q)\}$. Then $L(M) = \{a^n | n \geq 1\}$. See the Figure 16.

Now we will use M to construct the NFA N on the alphabet $\{a, b\}$ with the states $\{p, q, q', r', s'\}$ such that $F = \{s'\}$. Define δ as the following relation:

$$\{(p, a, q), (q, a, q)\} \cup \{(q, b, q'), (q', a, s'), (q, a, r'), (r', b, s')\}.$$

Then $L(M) = \{a^n ba | n \geq 1\} \cup \{a^n ab | n \geq 1\}$. See the Figure 17. Note that this machine is the result of gluing the previous machine M to a machine similar to the machine of the Example 2.6, in the node $q = p'$.

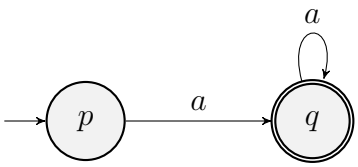

Figure 16: Example 2.9

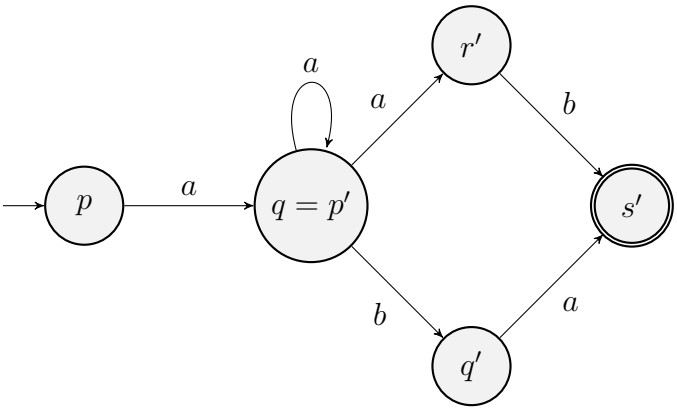

Figure 17: Example 2.9

Exercise 2.10. Let M and N be two NFA's that recognize the languages L_1 and L_2, respectively. Is there an NFA to recognize the language $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$?

Example 2.11. Consider the NFA M on the alphabet $\{a, b, c\}$ with the states $\{p, q, r\}$ such that $F = \{p\}$. Define δ as the following relation:

$$\{(p, a, q), (q, b, p), (p, a, r), (r, c, p)\}.$$

Then $L(M) = \{w_1 w_2 \dots w_n | w_i \in \{ab, ac\}, n \geq 0\}$. See the Figure 18. Note that this machine is the result of gluing the initial and the final states of the machine of the Example 2.6.

Exercise 2.12. Let M be an NFA that recognizes the language L. Is there an NFA to recognize the language $L^* = \{w_1 w_2 \dots w_n | n \geq 0, w_i \in L\}$?

2.2 The Equivalence Theorem

Theorem 2.13. *A language is accepted by an NFA if and only if it is regular. Equivalently, a language is accepted by an NFA if and only if it is accepted by a DFA.*

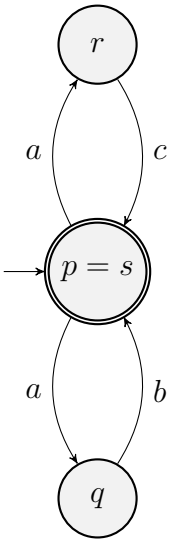

Figure 18: Example 2.11

Proof. Obviously, any language accepted by a DFA is also accepted by an NFA. Conversely, let $L = L(M)$, where M is an NFA with the transition relation δ, set of states $Q = \{q_1, \ldots, q_m\}$, and the set of final states F. We will construct a DFA N such that $L(N) = L(M)$. The idea is that the individual states of N will be the subsets of M denoted by $Q = \{Q_1, Q_2, \ldots, Q_{2^m}\}$, where in particular $Q_1 = \{q_1\}$ is to be the initial state of M. The set G of final states of N is given by:

$$G = \{Q_i | Q_i \cap F \neq \emptyset\}$$

The transition function σ of N is then defined by:

$$\sigma(Q_i, s) = \{r | \exists q \in Q_i \ \delta(q, s, r)\}$$

It is easy to see that N accepts exactly the strings for which there exists an accepting path in M. Hence, $L(M) = L(N)$. □

Remark 2.14. Simulating the parallel nature of non-determinism by some deterministic moves usually comes at a huge price. For instance, note that in the previous theorem, the deterministic simulator is exponentially bigger than the original non-deterministic machine.

2.3 Regular Operations

In this section we will explain some high-level methods to construct different regular languages, without the need to describe the actual machine every

time. In fact, we will present a descriptive criterion that characterizes all regular languages based on their common description form.

Lemma 2.15. *For any NFA M, there exists an equivalent non-restarting NFA N. Non-restarting means that the machine has no path from one of its final states to its start state.*

Proof. Add one new state q to M as the new start state and add an edge from q to r with label $s \in \Sigma$ if there is already an edge from the starting state of M to r with the same label. It is easy to see that N is non-restarting and equivalent to M, i.e., $L(M) = L(N)$. □

Theorem 2.16. *Let L_1 and L_2 be regular languages. Then all of the languages $\Sigma^* - L_1$, $L_1 \cup L_2$, $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$ and $L_1^* = \{w_1 w_2 \dots w_n | w_i \in L_1, n \geq 0\}$ are regular.*

Proof. Let M and N be two DFA's accepting the languages L_1 and L_2, respectively. To show the mentioned languages are regular, we describe the construction of an accepting NFA K for each case. Checking that these machines work is easy and left to the reader.

- For $\Sigma^* - L_1$, use M with the complemented final states, i.e., define $F_K = Q_M - F_M$.
- For $L_1 \cup L_2$, put M above N, add one new start state q mimicking the out edges of both start states of M and N. Note that this new machine can be non-deterministic.
- For $L_1 \cdot L_2$, put N after M, connecting all final states of M to some states of N mimicking the in-edges of the start state of N.
- For L_1^*, use M but for any edge that gets into a final state in M, draw a similar edge with the same source to the initial state of M. For this machine to work, M should be non-restarting and we can assume that condition for M, thanks to the Lemma 2.15.

□

Any combination of these operations on the singletons from Σ is called a regular expression. Note that any regular expression describes a regular language by the Theorem 2.16 and the Exercise 2.5.

Example 2.17. Using the convention of denoting $\{a\}$ by a, and omitting the points in concatenations, the regular expressions a^*, $(a \cup b)^* - a^*$, $(aa^*)(bb^*)$ and $(ab)^*$ describe the languages $\{a^n | n \geq 0\}$, $\{w \in \{a,b\}^* | w \text{ has at least one } b.\}$, $\{a^n b^m | m, n \geq 1\}$ and $\{(ab)^n | n \geq 0\}$, respectively.

The next theorem provides the converse of the previous statement. It can be read as a connection between the computability via certain models and being described by a certain form.

Theorem 2.18. *(Kleene) A language is regular iff it has a regular description via regular expressions*

Proof. One side is proved. For the converse, we will explain the idea behind the proof. First we have to introduce a generalized version of NFA's called GNFA's, with the same structure but this time using regular expressions as the labels of the edges. Acceptance in these machines is defined in a natural way. The machine reads the input, but not letter by letter. If it is in the state q and read the input up to w_i in $w = w_1 w_2 \dots w_n$, it can read any segment onwards such as $w_{i+1} \dots w_j$ and check if this segment is in the language of the label of an eadge from q to some other state, say q'. If yes, it can go to q' and jump on the input to w_{j+1}. It is clear that the usual DFA's are the special case of these machines. There are two thing to prove. First, one should prove that these generalized machines only accepts regular languages and then to show that any GNFA is equivalent (accepting the same language) to a smaller GNFA. This procedure reduces a usual DFA (read as a GNFA) to a single regular expression which completes the proof. □

3 Pumping Lemma

Consider the language $L = \{a^n b^n | n \geq 0\}$ over the alphabets $\{a, b\}$. Is L regular? Let us assume its regularity for a moment and try to construct a DFA to accept it. How should this machine work? The natural candidate for the algorithm is the following: First read all the a's till you reach the first b and remember the numbers of a's that you have read. Then do the same for b's, meaning that you have to read all the block of b's and remember their number again. Then, if you read an a again, reject, otherwise check whether the number of a's and b's that you have read so far are equal or not. The problem with this very natural algorithm is that it goes far beyond the power of the DFA's. The reason is simple: DFA's are not allowed to write and hence they do not have a reasonable memory (except maybe for a fixed finite amount of data, encoded through the finite states of the machine) and hence they can not remember the variable number of a's or b's that they have read in the algorithm. This lack of memory limits the power of DFA's dramatically. In this section we will explain how to use this weakness of DFA's to show that a given language is not regular. The main idea is the following: If you have a bounded memory, you will repeat yourself, eventually.

Theorem 3.1. *(Pumping Lemma). Let $L = L(M)$, where M is a DFA with n states. Let $x \in L$, where $|x| \geq n$. Then we can write $x = uvw$, where $|v| \neq 0$, $|uv| < n$ and $uv^i w \in L$ for all $i \geq 0$.*

Proof. Since x consists of at least n symbols, x must go through at least n state transitions as it scans x. Including the initial state, this requires atleast $n+1$ (not necessarily distinct) states. But since there are only n states in all, we conclude that M must be in at least one state more than once. Let q be the first state in which M finds itself at least twice. Then we can write $x = uvw$, where $\delta^*(q_1, u) = q$, $\delta^*(q, v) = q$, $\delta^*(q, w) \in F$. That is, M arrives in state q for the first time after scanning the last (right-hand) symbol of u and then again after scanning the last symbol of v. Since this "loop" can be repeated any number of times, it is clear that $\delta^*(q_1, uv^i w) = \delta^*(q_1, uvw) \in F$. Hence $uv^i w \in L$. Note that since q is the first repeated state $|uv| < n$, because if $|uv| \geq n$, we can write the same argument to find a repeating state in the run of M on uv. □

Example 3.2. The language $L = \{a^n b^n | n \geq 0\}$ is not regular. Assume otherwise. Then there exists M such that $L(M) = L$. Set $|Q_M| = n$. Then by the pupming lemma, since the length of $x = a^n b^n$ is bigger than n, it has a partition uvw such that $|uv| < n$. Therefore, v only consists of a's and since $|v| \neq 0$, the number of a's and b's in uv^2w can not equal which menas $uv^2w \notin L$. The same atgument also works for the language $L = \{x \in \{a, b\}^* | N_a(x) = N_b(x)\}$, where $N_a(x)$ and $N_b(x)$ are the number of a's and b's in x.

Exercise 3.3. Show the language $L = \{ww^R \in |w \in \{a, b\}^*\}$ is not regular, where w^R is the reverse of w, i.e., the word w written from right to left. For instance, $(abb)^R = bba$.

Example 3.4. The language $L = \{1^p | p \text{ is a prime number}\}$ on the alphabet $\{1\}$ is not regular. Assume otherwise. Then there exists M such that $L(M) = L$. Set $|Q_M| = n$ and pick $p \geq n$. Then by the pupming lemma, since the length of $x = 1^p$ is bigger than n, it has a partition uvw such that $|v| \neq 0$. Since the alphabet consists only of one element, any string can be identified with its length. Set $|u| = a$, $|v| = b$ and $|w| = c$. Therefore, $a + ib + c$ for any $i \geq 0$ is prime. Put $i = 0$, then $a + c$ is prime. Now put $i = a + c$, then $a + ib + c = (a + c)(1 + b)$. We know that $1 + b \geq 2$ and $a + c$ is prime. Therefore, their product can not be a prime. Hence $uv^{a+c}w \notin L$.

Exercise 3.5. Show that the language $L = \{1^{n^2} | n \geq 0\}$ on the alphabet $\{1\}$ is not regular. Interpreting $\{1\}^*$ as $\mathbb{N}$, it means that the set of all perfect squares is not regular.

Exercise 3.6. Show that the language $L = \{1^{2^n} | n \geq 0\}$ on the alphabet $\{1\}$ is not regular. It means that the set of all powers of two is not regular.

4 Turing Machines

Let us recall our informal notion of computation as a mechanical manipulation of symbols via the local operations of reading, writing and erasing, all governed by a finite set of given rules. In the previous sections we studied the limited power of computability when only reading is allowed. Now, we are ready to talk about the full story.

Definition 4.1. A Turing machine M is described by a tuple $M = (Q, \Sigma, \delta, q_1, F)$ containing:

- A finite set Q of possible states of M. We assume that Q contains a designated start state q_1, and $F \subseteq Q$ as the *halting* states.
- A finite set Σ of the input alphabet such that $\square \notin \Sigma$ where $\square$ is a symbol for blank,
- A transition function $\delta : (Q - F) \times (\Sigma \cup \{\square\}) \to Q \times (\Sigma \cup \{\square\}) \times \{L, R\}$.

We also need a formal version of *a piece of paper* on which the computation takes place. For that matter, the Turing machine $M = (Q, \Sigma, \delta, q_1, F)$ uses an unbounded linear tape, divided into infinite discrete cells. Initially, M receives its input $w = w_1 w_2 \dots w_n \in \Sigma^*$ on the tape with the rest of the tape filled with the blank symbol $\square$. The machine also has a read/write head. By convention, we assume that the head always starts on the leftmost square of the input. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function, meaning that if the machine is in the state p reading the letter s, it goes to the state q, changes the content of the cell to s' and go one cell right or left according to D where $\delta(p, s) = (q, s', D)$ and $D \in \{L, R\}$. The computation continues until it enters one of the halting states. If it does not occur, M goes on forever.

Just like a finite automaton, we can also represent a Turing machine by a directed graph with the nodes representing the states and edges with labels $s \mapsto s', D$ where $s, s' \in \Sigma \cup \{\square\}$ and $D \in \{L, R\}$, representing the transition function. If the transition function sends p to q reading the letter s with the instruction to erase s and write s' and move the head to D, there will be

an edge from p to q with the label $s \mapsto s', D$. Similar to what we had for automata, the start state is shown by a node with a small unlabelled in-edge and the halting states are indicated by double circles.

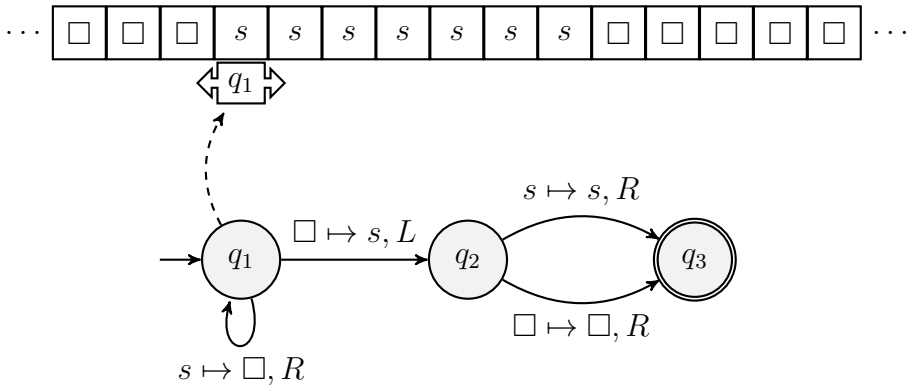

Figure 19: A TM with the states $\{q_1, q_2, q_3\}$, alphabet $\{s\}$ and $F = \{q_3\}$.

Definition 4.2. A partial function[1] $f : (\Sigma^*)^k \to \Sigma^*$ is *computable* by the Turing machine M, if for any $(w_1, w_2, \ldots, w_k) \in (\Sigma^*)^k$, the machine halts on the input $w_1 \square w_2 \square \ldots \square w_k$ iff $(w_1, \ldots, w_k)$ is in the domain of the function f. Moreover, when it halts, the value of f must appear on the tape between two lines of blanks while the head is on its first letter. A partial function is called computable if it is computable by some Turing machine.

To represent the truth values over any alphabet set Σ, fix an element $a \in \Sigma$ and represent **0** and **1** by the empty string and the string a, respectively.

Definition 4.3. A relation $R \subseteq (\Sigma^*)^k$ is called *semi-decidable* or *computably enumerable*[2], c.e. for short, if there exists a Turing machine M such that if $\vec{w} \in R$ it halts on the input $w_1 \square w_2 \square \ldots \square w_k$ and outputs **1** and if $\vec{w} \notin R$, it does not halt. It is called *decidable* or *computable* if there exists a Turing machine M such that it always halts on the input $w_1 \square w_2 \square \ldots \square w_k$ and outputs **1** if $\vec{w} \in R$ and **0** if $\vec{w} \notin R$.

[1]Unlike the usual practice of mathematics, when we write $f : A \to B$ we mean a partial function with $dom(f) \subseteq A$.

[2]The terminology here may seem a bit mysterious, simply because the machine is apparently implementing a sort of decision procedure and not enumeration. We will explain the reason behind this terminology in the following sections.

Remark 4.4. Note that the relation R is decidable iff its characteristic function χ_R:

$$\chi_R(\vec{w}) = \begin{cases} \mathbf{1} & \vec{w} \in R \\ \mathbf{0} & \vec{w} \notin R \end{cases}$$

is computable and it is c.e. if the function χ'_R with the domain R and the constant value $\mathbf{1}$ is computable.

4.1 Some Examples

In this subsection we will present some basic examples of Turing machines. We will first focus on the computable functions to see how to compute a concrete function. Then we will move to the decision procedures and their corresponding decidable and c.e. relations.

4.1.1 Computable Functions

Example 4.5. Consider the Turing machine M on the alphabet Σ with the states $\{p, q\}$ such that $F = \{q\}$. Define δ as $(p, \square, R)$ for the input (p, l) for any $l \in \Sigma$ and $(q, \square, R)$ for $(p, \square)$. Then M erases its input. The Figure 20 shows M for $\Sigma = \{a, b\}$. For $\Sigma = \{1\}$, it computes the constant zero function.

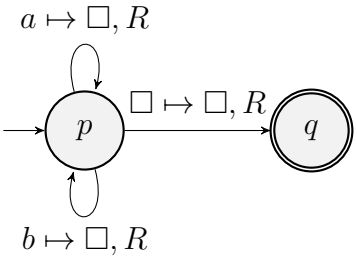

Figure 20: Example 4.5

Exercise 4.6. Let $u \in \Sigma^*$ be a fixed string. Describe a Turing machine that computes the function C_u defined by $C_u(w) = u$ for any $w \in \Sigma^*$.

Example 4.7. Consider the Turing machine M on the alphabet Σ with the states $\{p, q, r\}$ such that $F = \{r\}$. Pick $a \in \Sigma$ and define δ as (p, l, R) for the input (p, l) for any $l \in \Sigma$; (q, a, R) for $(p, \square)$; (q, l, L) for (q, l) for any $l \in \Sigma$ and $(r, \square, R)$ for $(q, \square)$. Then M computes the total function $f : \Sigma^* \to \Sigma^*$ that sends w to wa. The Figure 21 shows M for $\Sigma = \{a, b\}$. For $\Sigma = \{1\}$, it computes the successor function.

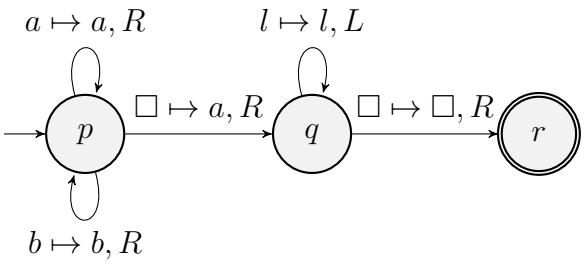

Figure 21: Example 4.7

Exercise 4.8. Let $u \in \Sigma^*$ be a fixed string. Describe a Turing machine that computes the function S_u defined by $S_u(w) = wu$ for any $w \in \Sigma^*$.

Example 4.9. Consider the Turing machine M on the alphabet Σ with the states $\{p, q, r, s\}$ such that $F = \{s\}$. Define δ as (p, l, R) for the input (p, l) for any $l \in \Sigma$; $(q, \square, L)$ for $(p, \square)$; $(r, \square, R)$ for (q, l) for any $l \in \Sigma \cup \{\square\}$; (r, l, L) for (r, l) for any $l \in \Sigma$ and $(s, \square, R)$ for $(r, \square)$. This machine erases the right-most letter of $w \in \Sigma^*$, if there is any, otherwise, it returns ϵ. Therefore, M computes the function:

$$\begin{cases} Pred(\epsilon) = \epsilon \\ Pred(wa) = w \end{cases}$$

The Figure 22 shows M for $\Sigma = \{a, b\}$. For $\Sigma = \{1\}$, it computes the predecessor function.

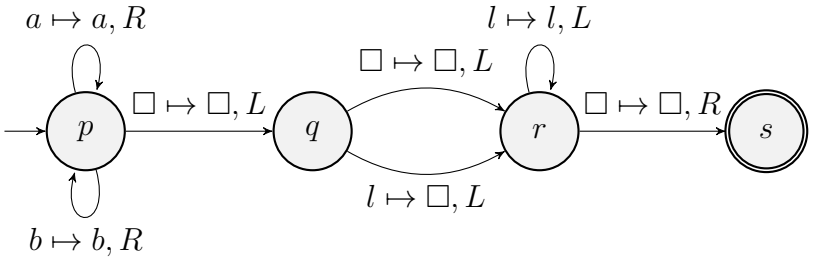

Figure 22: Example 4.9

As we can observe even with these very simple examples, spelling out all the details of a Turing machine and drawing its graph representation can be very complicated and time-consuming. Therefore, from now on, we will explain the *algorithm* behind the machine in words and in a more high-level manner. It helps to see the main ideas of the algorithm rather than the

details of the implementation. However, one has to be aware that every high-level operation must be reducible to the basic operations of the Turing machines, controlled by their finite number of states.

Example 4.10. (Projection) Consider the Turing machine `Projection` on any set of alphabets Σ:

(I) Go right and erase everything till you see the $(i-1)$th blank. Then go right but do nothing till you see the next blank. By investigation whether these two blanks are adjacent or not you can find out whether the data between these two blanks is EMPTY or NON-EMPTY. Then again go right and erase the tape till you read the kth blank.

(II) If EMPTY, halt. If NON-EMPTY, go left till you read a letter other than the blank. Keep going till seeing another blank. Go one step right and halt.

This machine reads the input $w_1\square w_2\square \dots \square w_k$ and outputs w_i for $i \leq k$. Therefore, it computes the projection function $I^i_k(w_1, \dots, w_k) = w_i$.

Example 4.11. (Moving Right) Consider the Turing machine `MovingRight` on any set of alphabets Σ:

(I) If you read blank, go one step right and halt. If you read any letter l go right till you reach a blank,

(II) Go one step left, read the content. If it is in Σ, remember it, change it to blank. Go one step right, write the letter, go left and go to (II). If it is blank, go two steps right and halt.

This machine move the input string one step to right.

Example 4.12. (Concatenation) Consider the Turing machine `Concatenation` that reads $u\square v$ and then apply the `MovingRight` algorithm as in the Example 4.11. Then it moves u, one step to right to produce the concatenation uv. Therefore, the machine computes the concatenation function $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ sending (u, v) to uv. Note that if the algorithm `MoveRight` reads $u\square v$, it does not care what is in the right hand side of the middle blank. The `Concatenation` uses this property, substantially. Note that for the language $\Sigma = \{1\}$ and identifying $\{1\}^*$ by $\mathbb{N}$, the function f is the usual addition.

Example 4.13. (Sequence Concatenation) Consider the Turing machine `SeqConcatenation` that reads $u_1\square u_2\square \dots \square u_n$ where u_n is not empty and then:

(I) Run `MovingRight` as in the Example 4.11.

(II) Go right and check whether you see the first blank in the form $\square\square$ or not. If yes, go left till seeing the blank. Go right and halt. If no, go to the start of the input and go to (I).

This machine computes the concatenation $u_1u_2\dots u_n$.

Example 4.14. (Copy) Consider the following Turing machine `Copy` on any set of alphabets Σ:

(I) If you read blank go right and halt. If you read any letter l, remember it, erase it, go right till reaching the first blank. Then write l. Run `MoveRight`. Go right till reaching a blank. Write l again. Run `MoveRight`. Go left till seing the third blank. Go right. Go to (I).

This machine sends w to $w\square w$.

Example 4.15. (Multiplication) Consider the Turing machine `Multiplication` on any set of alphabets Σ:

(I) If the content of your current cell or the two cells in the right is blank, erase everything and halt. Otherwise, go to the first of the input and go to (II).

(II) If you read blank, go to (III). If you read any letter l other than blank, erase it and then go right till you reach the first blank. Go one step further right, run `Copy`. Go left till reading the second blank. Go one step right. Go to (II).

(III) Go two steps right. Run `SeqConcatenation`.

This machine computes $f : \Sigma^* \times \Sigma^* \to \Sigma^*$ that sends (u, v) to $v^{|u|}$. Note that for the language $\Sigma = \{1\}$ and identifying $\{1\}^*$ by $\mathbb{N}$, the function f is the usual multiplication.

Example 4.16. (Boolean Operations) Consider the Turing machine `Disjunction` defined as follows: Run `Concatenation`. Scan the output. If it consists of two a's, erase one a, move the head over that a and halt. Otherwise, just halt. This machine computes the disjunction function $\vee : \{\mathbf{0}, \mathbf{1}\} \times \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$. For the conjunction function $\wedge : \{\mathbf{0}, \mathbf{1}\} \times \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$ use `Multiplication`. For negation, $\neg : \{\mathbf{0}, \mathbf{1}\} \to \{\mathbf{0}, \mathbf{1}\}$, use the following `Negation` machine: Scan the input. If it consists of one a, erase the a and halt. If the input is empty, write one a and halt. Otherwise, just halt.

Exercise 4.17. Show that the following function $C : \{\mathbf{0}, \mathbf{1}\} \times \Sigma^* \times \Sigma^* \to \Sigma^*$ is computable:

$$C(u, v, w) = \begin{cases} v & u = \mathbf{0} \\ w & u = \mathbf{1} \end{cases}$$

Exercise 4.18. Let $S \subseteq \Sigma^*$ be a finite set and $f : S \to \Sigma^*$ be any function. Show that f is computable. Note that it implies that any function with finite domain is computable.

4.1.2 Computably Enumerable and Decidable Relations

Example 4.19. Consider the Turing machine M on the alphabet $\Sigma = \{1\}$ with the states $Q = \{p, q, r, s\}$ such that $F = \{s\}$. Define δ as $(q, 1, R)$ for the input $(p, 1)$; (q, l, R) for the input (q, l) for any $l \in \{1, \square\}$; $(r, 1, R)$ for $(p, \square)$; (s, l, L) for (r, l) for any $l \in \{1, \square\}$. The machine halts if it reads the empty string and outputs 1 but if we feed the machine any other string, then it goes right forever. Therefore, M computes the partial function $f : \{1\}^* \to \{1\}^*$ with domain $\{\epsilon\}$ and the value $\mathbf{1} = 1$. In other words, it shows that the set $\{\epsilon\}$ is computable enumerable. See the Figure 23.

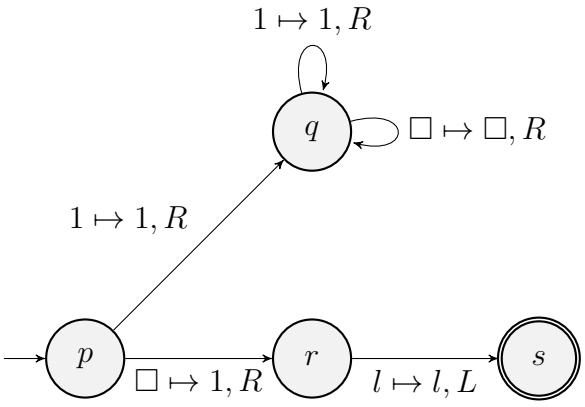

Figure 23: Example 4.19

Exercise 4.20. By constructing two Turing machines, show that the equality relation $\{(u, v) \in \Sigma^* \times \Sigma^* | u = v\}$ is both decidable and computably enumerable.

Exercise 4.21. By constructing two Turing machines, show that the first segment relation $\{(u, v) \in \Sigma^* \times \Sigma^* | \exists w(uw = v)\}$ is both decidable and computably enumerable.

Example 4.22. Any regular language L is also decidable. Let us assume that the DFA M recognizes L. We will construct a Turing machine N deciding L. The algorithm N first runs M till it reads all the input. If it accepts, N erases everything, writes **1** and halts. If it rejects, N erases everything, writes **0** and halts.

The previous example shows that the class of all decidable languages extends the class of all regular languages. This extension is proper as we will see in the following example:

Example 4.23. Consider the following Turing machine:

(I) If you read blank, write **1** and halt. If you read b, erase everything and right **0**. If you read a, erase it and then go right till you reach the blank. Go right and check whether the last letter is b or not. If not, erase everything, write **0** and halt, if yes, erase the last letter and go left till reaching the blank. Go right and go to (I).

This machine computes the function $f : \{a, b\}^* \to \{\mathbf{0}, \mathbf{1}\}$ that sends all strings of the form $a^n b^n$ to **1** and the rest to **0**. Therefore, it decides the language $L = \{a^n b^n | n \geq 0\}$ and hence there are some decidable non-regular languages.

Exercise 4.24. Show that the language $\{1^{2^n} | n \geq 0\}$ over the alphabet $\{1\}$ is decidable.

Example 4.25. Any decidable language is also computably enumerable. Let M be the decision procedure for R. Then define the machine N as first ruining M, if it outputs **1** halt. If it outputs **0**, go to right forever.

Remark 4.26. The inclusion of the Example 4.25 is also proper. We will provide a c.e. but undecidable language later in the course.

5 The Variants of Turing Machines

In the previous section we introduced the deterministic Turing machines with one linear unbounded tape, and we have defined the notion of computability based on that definition. Fortunately, the notion of computability is extremely *robust* and independent from the implementing details of the Turing machines. In the following we will mention some variants of the Turing machines with different structural details but the same power of computation:

- Usually Turing machines are defined without the set F and the transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$ as a *partial function*. In these machines, halting means reaching a configuration outside of the domain of δ.

- Just like the case for NFA's, it is also possible for Turing machines to be *non-deterministic*. The idea, again, is allowing the function δ to be a relation.

- For computably enumerable and decidable relations, sometimes people use the machines with two specified states called *the accept state* and *the reject state*. If the machine reaches the accept state (reject state) it accepts (rejects) the input, otherwise the machine does not halt on the input. In these machines, accepting or rejecting the language has nothing to do with the outputs of the machine.

- It is possible to use a greater set of *working alphabets* than the the set of the *input alphabets* Σ. This set is usually denoted by Γ and it should be an extension of Σ.

- The machines can have access to *more local moves* than the two basic moves of left and right. For instance in some cases it is useful to have a machine with three moves Left, Right and *Stay*.

- It is also possible to change the details of *the tape* of the machine. The machine could have only *one semi-unbounded tape* that is unbounded only on one direction and bounded on the other or it can have *fixed number of tapes with separate heads for any tape* or even *a plane* type of tape with one head with four moves for four directions.

- The machine can recieve its input in many different ways. For instance, the head can be on its right-most end of the input or when the machine has a one-sided tape, the head can be on the first cell of the bounded side of the tape.

6 A Bit of Recursion Theory

As we have observed for finite automata and their relation with regular expressions, in some cases, it is possible to develop a machine independent characterization of computability with respect to a given notion of computation. Turing machines are no exception in this regard. In this section we will introduce a machine-independent characterization of computable functions

and computably enumerable and decidable relations. As we can expect, such a characterization is helpful in proving the computability of functions and relations without providing the actual possibly complex Turing machines. However, we have to pay the price for handling these complicated details at some point and this section and its equivalence theorem may be the reasonable point to handle them. However, we have to confess that in this section we will only use the computability in its very high-level sense, avoiding the implementation details, altogether. Admittedly, this is a sort of cheating, but in our defence, let us emphasize that for such a short lecture, it may be reasonable to explain the main ideas behind the *algorithms* rather than getting lost in the implementation details.

Theorem 6.1. *(Basic Functions) The basic functions $Z(w) = \epsilon$, $S_a(w) = wa$ for $a \in \Sigma$ and $I^i_k(w_1, w_2, \ldots, w_k) = w_i$ for $1 \leq i \leq k$ are all computable.*

Proof. See Example 4.5, Example 4.7 and Example 4.10. □

Theorem 6.2. *(Composition) If $f : (\Sigma^*)^k \to \Sigma^*$ and $g_i : (\Sigma^*)^l \to \Sigma^*$ for $1 \leq i \leq k$ are computable, then so is $h = f(g_1, \ldots, g_k)$.*

Proof. Use k many tapes for the computation of h. First for each $i \leq k$, run g_i on the ith tape, one after another. If one of them does not halt, then as we expect, the new machine does not halt as well. If all halts with the outputs $g_i(\vec{w})$, then copy all the elements of all the tapes in the first tape as $g_1(\vec{w})\Box g_2(\vec{w})\Box \ldots \Box g_k(\vec{w})$ and run f on the first tape. □

Application 6.3. (Substitution) Let $R \subseteq (\Sigma^*)^k$ be a c.e. relation and for any $1 \leq i \leq k$, the function $g_i : (\Sigma^*)^l \to \Sigma^*$ be computable. Then the relation $S = R(g_1, \ldots, g_k)$ is also c.e. If R is decidable and all g_i's are total, then S is also decidable.

Proof. It is a consequence of the closure of the class of computable functions under composition and the fact that $\chi'_S = \chi'_R(g_1, \ldots, g_k)$. Note that if R is decidable, then by totality of g_i's, we have $\chi_S = \chi_R(g_1, \ldots, g_k)$ and hence the result follows. □

Theorem 6.4. *(Primitive Recursion) Assume the functions $g : (\Sigma^*)^k \to \Sigma^*$ and for any $a \in \Sigma$, $h_a : \Sigma^* \times (\Sigma^*)^k \times \Sigma^* \to \Sigma^*$ are computable. Then the function $f : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ with the following definition is also computable:*

$$\begin{cases} f(\epsilon, \vec{v}) = g(\vec{v}) \\ f(ua, \vec{v}) = h_a(u, \vec{v}, f(u, \vec{v})) \end{cases}$$

Proof. First note that to compute f on any input in the form $(wa, \vec{v})$, it is enough to know the value $f(w, \vec{v})$. Using this observation, we can develop a possible computing procedure. We reduce computing f on some w to computing f on a shorter w and we keep following this procedure till reaching $f(\epsilon, \vec{v})$ which is computable by $g(\vec{v})$. Now, the strategy is following the procedure that we have constructed, backwardly. First compute $f(\epsilon, \vec{v})$, then compute f for the previous element using the appropriate h_a, till you reach the input u. More formally: Use a 4-taped machine to compute f. Write u on the first tape and $\vec{v}$ on the third tape. First copy $\vec{v}$ on the fourth tape, run g on $\vec{v}$. Then read the elements of u one by one and in each step, copy u on the second tape, keep the part of u that you have read and erase the rest, and apply the predecessor. If you have read a, copy the content of the second and the third tape to the fourth tape in the order second, third, fourth and then apply h_a on the fourth tape. Erase the second tape and do it again till you scan all the elements of u. Then halt and output the content of the fourth tape. □

Example 6.5. The concatenation function $Con(u, v) = uv$ is computable. Use recursion on v to define $Con(u, v)$:

$$\begin{cases} Con(u, \epsilon) = I_1^1(u) \\ Con(u, va) = S_a(Con(u, v)) \end{cases}$$

Since both I_1^1 and S_a are computable, the so is Con.

Exercise 6.6. As usual, identify the set of natural numbers with $\{1\}^*$. Then show that the functions $Sum(m, n) = m+n$, $Prod(m, n) = mn$, $Exp(m, n) = m^n$, $Fact(n) = n!$, the predecessor $Pred(n) = max\{0, n-1\}$, the proper subtraction $PSub(m, n) = max\{0, m-n\}$, the division $div(m, n) = \lfloor \frac{m}{n+1} \rfloor$ and the remainder $r(m, n) = m - (n+1)div(m, n)$ are all computable.

Exercise 6.7. Show that over the alphabets $\{1\}^*$, the function $Sign(n)$ with the following definition:

$$Sign(n) = \begin{cases} 1 & n = 0 \\ 0 & n \geq 1 \end{cases}$$

is computable. Then use $Sign$ and $PSub$ to prove that the inequality relation $\{(m, n) \in \{1\}^* \times \{1\}^* | m \leq n\}$ is decidable. Finally show that the equality relation and the strict inequality are also decidable.

Definition 6.8. Any function constructed from the basic functions and the operations of composition and primitive recursion is called *primitive recursive*.

Application 6.9. (Boolean Combinations) Let $R, S \subseteq (\Sigma^*)^k$ be computably enumerable relations. Then so is $R \cap S$. If R is also decidable, then so is R^c. Therefore, all boolean combinations of decidable relations are decidable.

Proof. For the first part, note that $\chi'_{R\cap S} = C_{\mathbf{1}}(\chi'_R, \chi'_S)$ where $C_{\mathbf{1}} : \Sigma^* \times \Sigma^* \to \{\mathbf{1}\}$ is the constant function, constructible via composition and basic functions as $C_{\mathbf{1}}(u, v) = S_{\mathbf{1}}(Z(I^1_2(u, v)))$. Therefore, by the closure under composition we see that if χ'_R and χ'_S are computable, then so is $\chi'_{R\cap S}$.
For the second part, first define the following function $\neg' : \Sigma^* \to \{\mathbf{0}, \mathbf{1}\}$ by recursion as $\neg'(\epsilon) = \mathbf{1}$ and $\neg'(wa) = \epsilon$. Note that the restriction of $\neg'$ to $\{\mathbf{0}, \mathbf{1}\}$ is the usual negation. Then the rest is a consequence of the closure under composition and the fact that $\chi_{R\cap S} = Con(\chi_R, \chi_S)$ and $\chi_{R^c} = \neg'(\chi_R)$ where Con is concatenation. □

Remark 6.10. The complement of a c.e. relation is not necessarily c.e. In the last section we will see an interesting counter-example.

Theorem 6.11. *(Union) Let $R, S \subseteq (\Sigma^*)^k$ be computably enumerable relations. Then so is $R \cup S$.*

Proof. Let M and N be the algorithms for R and S. Then for an algorithm K for $R \cup S$, run M and N in parallel. More formally, use a 2-taped Turing machine and copy the input on the second tape as well. Then run one step of M on the first tape and then one step of N on the second tape and keep going alternately. The whole process halts if at least one of M or N halts and it outputs $\mathbf{1}$ when it halts. This algorithm clearly computes $\chi'_{R\cup S}$. Note that at the first glance, it can be really tempting to simplify the algorithm by running M first and then applying N. The problem with this algorithm is that it ignores the possibility in which M does not halt while N halts on the input. So the algorithm get stuck in running M and can not see that N actually halts on the input. □

Application 6.12. (Definition by cases) Let $R \subseteq (\Sigma^*)^k$ be a decidable relation and $g, h : (\Sigma^*)^k \to \Sigma^*$ be two computable functions. Then the function $f : (\Sigma^*)^k \to \Sigma^*$ defined as:

$$f(\vec{u}) = \begin{cases} g(\vec{u}) & \vec{u} \in R \\ h(\vec{u}) & \vec{u} \notin R \end{cases}$$

is also computable.

Proof. First note that the function $C : \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^*$ with the definition:

$$C(u, v, w) = \begin{cases} v & u = \epsilon \\ w & u \neq \epsilon \end{cases}$$

is computable. It is enough to use primitive recursion on u to define $C(\epsilon, v, w) = I_2^1(v, w)$ and $C(ua, v, w) = I_3^3(u, v, w)$. The rest of the theorem is a consequence of the closure under composition and the fact that we have $f(\vec{u}) = C(\chi_R(\vec{u}), h(\vec{u}), g(\vec{u}))$. □

Example 6.13. Let Σ be a set of alphabets. Then the length function $|\cdot| : \Sigma^* \to \{\mathbf{1}\}^*$ sending w to $\mathbf{1}^n$ where n is the length of w, is computable. It is enough to use recursion on u to define $|u|$ as $|\epsilon| = \epsilon$ and $|ua| = S_{\mathbf{1}}(|u|)$.

Lemma 6.14. *For any primitive recursive function over $\{1\}$, the function $\tilde{f}$ over Σ with the definition $\tilde{f}(\vec{u}) = \mathbf{1}^{f(|\vec{u}|)}$ is primitive recursive over Σ.*

Proof. The proof is easy and uses induction on the structure of f. □

Exercise 6.15. Show that both of the length-inequality relation

$$LIneq(u, v) = \{(u, v) \in \Sigma^* \times \Sigma^* |\ |u| \leq |v|\}$$

and the length-equality relation

$$Leq(u, v) = \{(u, v) \in \Sigma^* \times \Sigma^* |\ |u| = |v|\}$$

are decidable.

Exercise 6.16. First show that the set $\{\epsilon\}$ is decidable. Then prove that for any $a \in \Sigma$, the set $\{a\}$ is also decidable. We will denote this predicate with Eq_a.

Exercise 6.17. Show that the function $Seg(u, v)$ computing the first segment of u consisting of the leftmost $|v|$ elements of u, is computable. Then show that the $|v|$-th component function u_v computing the $|v|$'th letter of u (for $|u| < |v|$ answers ϵ) is computable.

Application 6.18. (Bounded Search) Let $f : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ be a total computable function. Then so is $\mu.|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$ where $\mu.|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$ is $\mathbf{1}^k$ where $k \leq |w|$ is the length of the shortest u such that $f(u, \vec{v}) = \mathbf{1}$. If there is no such u the output is $\mathbf{1}^{|w|+1}$.

Proof. Denote $\mu|u| \leq |w|\ [f(u, \vec{v}) = \mathbf{1}]$ by $g(w, \vec{v})$. We will use primitive recursion on w to show that g is computable. For $w = \epsilon$, check whether $f(\epsilon, \vec{v}) = \mathbf{1}$ or not. We can do it because $\{\mathbf{1}\}$ is decidable. Then using definition by cases, if $f(\epsilon, \vec{v}) = \mathbf{1}$, set $g(\epsilon, \vec{v}) = \mathbf{1}^0 = \epsilon$, otherwise, set $g(\epsilon, \vec{v}) = \mathbf{1}$. For $g(wa, \vec{v})$, check whether $|g(w, \vec{v})| = |w| + 1$ or not. We can do it because the length equality is decidable. If $|g(w, \vec{v})| \neq |w| + 1$, then define $g(wa, \vec{v}) = g(w, \vec{v})$, otherwise, check whether we have $f(wx, \vec{v}) = \mathbf{1}$ for some $x \in \Sigma$ or not. In the first case, define $g(wa, \vec{v}) = \mathbf{1}^{|w|+1}$, otherwise, set $g(wa, \vec{v}) = \mathbf{1}^{|w|+2}$. □

Remark 6.19. Note that for any decidable relation $R(u, \vec{v})$, the function $f(w, \vec{v}) = \mu|u| \leq |w|.R(u, \vec{v})$ is computable, where $\mu|u| \leq |w|.R(u, \vec{v})$ is $\mathbf{1}^n$ where $n \leq |w|$ is the length of the shortest possible u such that $R(u, \vec{v})$. And if there is no such u, then $\mu|u| \leq |w|.R(u, \vec{v})$ is $|w| + 1$. The reason is that $R(u, \vec{v})$ is nothing but $\chi_R(u, \vec{v}) = \mathbf{1}$.

Exercise 6.20. Show that the function $exp(m, n)$ computing the exponent of $m + 2$ in $n + 1$, meaning the greatest k such that $(m + 2)^k | n + 1$, is computable.

Example 6.21. The function $NextP(x)$ that finds the least prime $p > x$ is computable because it is definable via bounded search $NextP(x) = \mu y \leq 2x.[Prime(y) \wedge x < y]$. Note that we are using Bertrand's postulates that states between any number $x \geq 1$, there exists a primes number $x < p \leq 2x$. Moreover, it is also possible to use $NextP$ to define the function p_x that computes the x-th prime number. By recursion define $p_0 = SS(0) = 2$ and $p_{x+1} = NextP(p_x)$.

Exercise 6.22. Show that the function $ind(m, n)$ computing the exponent of p_m in $n + 1$ is computable.

Example 6.23. The function $GP(n)$ that computes m such that p_m is the greatest prime that $p_m | n + 2$, is computable because it is constructible via bounded search $\mu i \leq n + 2$. $[\exists k \leq n + 2(p_{n+2-i} k = n + 2)]$. Here we use the fact that if $p_m | n + 2$, then $m \leq p_m \leq n + 2$.

Example 6.24. The equality relation over the alphabet Σ is decidable because it has the following construction: $(|u| = |v|) \wedge |x_{(u,v)}| = |u| + 1$ where $x_{(u,v)} = \mu|w| \leq |u| \ \neg[u_w \equiv v_w]$ and $y \equiv z$ is an abbreviation for $\bigvee_{a \in \Sigma}(Eq_a(y) \wedge Eq_a(z))$.

Exercise 6.25. Let $f : (\Sigma^*)^k \to \Sigma^*$ be a computable function. Then the graph of f, i.e, $\{(\vec{x}, y) | f(\vec{x}) = y\}$ is a c.e. relation. If f is also total, its graph is also decidable.

Application 6.26. (Bounded Quantifiers) Let $R \subseteq \Sigma^* \times (\Sigma^*)^k$ be a decidable relation. Then so is $S(w, \vec{v}) = \forall|u| \leq |w| \ R(u, \vec{v})$ and $T(w, \vec{v}) = \exists|u| \leq |w| \ R(u, \vec{v})$.

Proof. By closure under booleans, it is enough to prove the theorem for T. By bounded search, $f(w, \vec{v}) = \mu.|u| \leq |w| \ [\chi_R(u, \vec{v}) = \mathbf{1}]$ is computable and total. Now check whether $|f(w, \vec{v})| = |w| + 1$ or not. In the first case, $\exists|u| \leq |w| \ R(u, \vec{v})$ does not hold. In the latter case, it holds. □

Example 6.27. The set of all prefect squares in the language $\{1\}^*$ is decidable because it is definable by

$$Square(x) = [\exists y \leq x(x = y^2)]$$

The latter is decidable because multiplication is total and computable and equality is decidable. Hence, by substitution the relation $x = y^2$ is decidable. Finally, since decidability is closed under bounded quantifiers we will have the claim.

Example 6.28. The set of all prime numbers in the language $\{1\}^*$ is decidable because it is definable by

$$Prime(x) = [(x > 1) \wedge \forall yz \leq x((x = yz) \rightarrow (y = x \vee y = 1))]$$

The latter is decidable because equality and inequality are decidable. Hence, by substitution the relations $x > 1$, $x = yz$, $y = x$ and $y = 1$ are decidable. Finally, since decidability is closed under boolean operations and bounded quantifiers we will have the claim.

Remark 6.29. Note that the languages of the previous examples, $\{1^{n^2} | n \geq 0\}$ and $\{1^p | p \text{ is prime}\}$ are not regular. Therefore, we can also use them to show that the inclusion of regular languages in decidable languages is proper.

Exercise 6.30. (Bounded Concatenation) Assume that $f : \Sigma^* \times (\Sigma^*)^k \rightarrow \Sigma^*$ is computable. Show that the function $\bar{f} : \Sigma^* \times (\Sigma^*)^k \rightarrow \Sigma^*$ with the following definition is computable, as well:

$$\bar{f}(u, \vec{v}) = f(\mathbf{1}^0, \vec{v}) f(\mathbf{1}^1, \vec{v}) \dots f(\mathbf{1}^{|u|}, \vec{v})$$

So far, we have shown that all primitive recursive functions are computable. The natural question is that whether these functions exhaust the whole class of computable functions? The answer is of course not. Because there are so many partial computable functions while all primitive recursive functions are total by construction. However, the partiality may not be an important problem. Therefore, let us revise the question: Is a total computable function primitive recursive? The answer is again negative:

Example 6.31. Consider the Ackerman function defined recursively by:

$$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & m > 0, n > 0 \end{cases}$$

This function is not primitive recursive because it grows faster than any primitive recursive function. More precisely, it is not hard to prove that for any primitive recursive function $f(\vec{x})$, there exists a number t such that for any $\vec{x}$, we have $f(\vec{x}) < A(t, max\{\vec{x}\})$. Therefore, if A is primitive recursive, there is t such that for any x, y we have $A(x, y) < A(t, max\{x, y\})$. Hence $A(t, t) < A(t, t)$. On the other hand, it is very clear that this function is computable. What is your strategy to compute it?

To capture the full power of computability and its partial nature, we need another operation. But before introducing that operator let us state a technical lemma and a theorem on the existential quantifiers. It helps to understand how the final operator works:

Lemma 6.32. *(Enumerability of the Total Space) There is a total computable function $en : \Sigma^* \to \Sigma^*$ whose restriction to $\{\mathbf{1}\}^*$ is surjective.*

Proof. First fix an order over the elements of $\Sigma = \{s_1, s_2, \ldots, s_m\}$. Use a 2-taped machine. The input is on the first tape. Now, for reading any letter from the input do the following: Check the second tape. Find the leftmost s_m. Change the content of the previous cell from blank or s_i to s_{i+1}, and change the content of everything in the right of this cell with content s_m to s_1. Keep iteratring this procedure till reading all the input. Then halt. This algorithm computes a total function and its application only on the elements of $\{\mathbf{1}\}^*$ is enough to cover Σ^*. For instance, for $\Sigma = \{s_1, s_2, s_3\}$, the following is the sequence of all $\{en(\mathbf{1}^n)\}_{n=0}^{\infty}$: ϵ, s_1, s_2, s_3, s_1s_1, s_1s_2, s_1s_3, s_2s_1, s_2s_2, s_2s_3, s_3s_1, ... □

Theorem 6.33. *(Unbounded Existential Quantifier) Let $R \subseteq \Sigma^* \times (\Sigma^*)^k$ be a c.e. relation. Then so is $S(\vec{v}) = \exists u R(u, \vec{v})$.*

Proof. Let M be the algorithm for R and we want to define an algorithm N for S. The idea is reading the input $\vec{v}$ and then computing $M(u, \vec{v})$ for all possible u's to check whether there exists a u such that $M(u, \vec{v}) = \mathbf{1}$ or not. For this purpose, first order all u's in a length increasing order u_0, u_1, ... Consider the following tempting search algorithm: Apply M on $(u_0, \vec{v})$ then on $(u_1, \vec{v})$ and so on, till one of them outputs $\mathbf{1}$, otherwise do not halt. The problem with this algorithm is that it is possible that $M(u_0, \vec{v})$ does not halt while $M(u_1, \vec{v}) = \mathbf{1}$. Therefore, the machine gets stuck in u_0 and never reaches the working u_1. To overcome this problem, we run all these algorithms in parallel and step by step in the following manner and we call the algorithm N. Let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective. In the stage n, N applies M on $(en(n_0), \vec{v})$ for n_1 many steps where $n + 1 = 2^{n_0}(2n_1 + 1)$. Then N checks whether it

reached a halting state or not. If it does, it also halts. Otherwise, it erases everything and go to the next stage. This algorithm clearly captures $S(\vec{v})$. The reason is the following: If N halts and outputs $\mathbf{1}$, then it means it halts in some stage, say n. Then it means that the machine M applied on the input $(en(n_0), \vec{v})$ halts after n_1 many steps. Hence, we have $R(en(n_0), \vec{v})$ which implies $S(\vec{v})$. Conversely, if $S(\vec{v})$, then there exists some string u such that $R(u, \vec{v})$. Then if M runs on $(u, \vec{v})$, it halts after say m many steps. Since the restriction of en to $\{\mathbf{1}\}^*$ is surjective, there exists k such that $en(k) = u$. Then the machine N on the stage n where $n + 1 = 2^k(2m + 1)$ simulates the work of M on $(u, \vec{v})$ and hence it halts and outputs $\mathbf{1}$. □

Application 6.34. (Bounded Existential Quantifier) Let $R \subseteq \Sigma^* \times (\Sigma^*)^k$ be a c.e. relation. Then so is $T(w, \vec{v}) = \exists |u| \leq |w| \; R(u, \vec{v})$.

Proof. Since $R(u, \vec{v})$ is c.e. and $|u| \leq |w|$ is decidable, then $|u| \leq |w| \wedge R(u, \vec{v})$ is also c.e. Hence, $\exists u(|u| \leq |w| \; R(u, \vec{v}))$ is c.e. □

Example 6.35. Let $p(\vec{x}, \vec{y})$ be a polynomial with non-negative integer coefficients. Then the relation $\exists \vec{y} p(\vec{x}, \vec{y}) = 0$ is computably enumerable, because all polynomials are compositions of addition and multiplications and hence computable. And then by decidability of the equality and by substitution, the relation $p(\vec{x}, \vec{y}) = 0$ is decidable and by closure of the c.e. relations under unbounded existential quantifiers we have the claim.

Example 6.36. Let $\mathcal{L}$ be a first-order language. Then the set of all $\mathcal{L}$-tautologies over the alphabets $\mathcal{L} \cup \{;\}$ is computably enumerable. Note that we use the symbol ";" to separate formulas and hence to represent the proofs as words in the new language $\mathcal{L} \cup \{;\}$. To show why this set is c.e., note that it has the following description: $\exists \pi Prf(\pi, A)$ where $Prf(\pi, A)$ is the relation "π is a proof of A". Intuitively, it is clear that checking whether something is a proof of some statement is decidable. We only need to check some basic syntactical properties of π to check whether it is really a proof or not.

Exercise 6.37. The domain and the range of any computable function is computably enumerable.

Theorem 6.38. *(Unbounded Search) Let the function $g : \Sigma^* \times (\Sigma^*)^k \to \Sigma^*$ be a computable function. Then the function $f : (\Sigma^*)^k \to \Sigma^*$ with the following definition is also computable:*

$$f(\vec{v}) = \mu |u|. \; [g(u, \vec{v}) = \mathbf{1}]$$

where $\mu|u|$. $[g(u,\vec{v}) = \mathbf{1}]$ *means the string* $\mathbf{1}^n$ *where* n *is the length of the shortest possible* u *such that* $g(u,\vec{v})$ *is defined and* $g(u,\vec{v}) = \mathbf{1}$ *and for any* w *shorter than* u*, the value* $g(w,\vec{v})$ *is defined and* $g(w,\vec{v}) \neq \mathbf{1}$*. If there is no such* u*,* $f(\vec{v})$ *is not defined.*

Proof. The most natural algorithm is computing $g(u,\vec{v})$ for the given $\vec{v}$ and all possible u's, one after another in a length increasing order, till we see the first output $\mathbf{1}$ and then we can compute the shortest u. The problem with this algorithm is that it ignores the situation in which the shortest possible length is n and we are checking u before w, both with length n while $g(u,\vec{v})$ is not defined and $g(w,\vec{v})$ is. To overcome this issue, we check all possible lengths but running the machine for g on $(u,\vec{v})$ for all u's with the same length, in parallel. More formally, let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective and it enumerates all strings in a length increasing order and M be the machine computing g. Then define the algorithm N in the following way. It consists of some stages that itself is partitioned to some steps. In the stage n, N first computes all $en(i)$'s, one after another, to find the first number N_m such that the length of $en(N_m)$ becomes n. This is possible because en enumerates the elements in a length increasing order. Then we know that all elements with length n are $en(N_n)$, $en(N_n+1)$ till $en(n_m+2^{n-1})$. Now in each step m, N first computes m_0 and m_1 where $m+1 = 2^{m_0}(2m_1+1)$ and then if $m_0 > 2^{n-1}$, the machine goes to the next step, otherwise N applies M on $(en(N_n+m_0),\vec{v})$ for m_1 many steps and then checks whether M reached a halting state or not. If it does, N check whether the output of M is $\mathbf{1}$ or not. If it is $\mathbf{1}$, it halts and outputs n. Otherwise, it erases everything, write m_0 somewhere to remember it and go to the next step. The machine goes to its next stage, if it gets the halting state of M with output not equal to $\mathbf{1}$ for all possible $0 \leq m_0 \leq 2^{n-1}$.
If there exists a shortest possible u such that $f(u,\vec{v}) = \mathbf{1}$, then M halts on any shorter input and outputs something different than $\mathbf{1}$. Hence, N passes all the stages before n, and in the stage n finally meets the answer u at some point since en is surjective. If there is no shortest u, then there is a number n such that M halts on any $(u,\vec{v})$ where $|u| < n$ and for u with length n at least at one point it does not halt and wherever it halts it does not output $\mathbf{1}$. Therefore, N passess all the stages before n but in the stage n since there is no u such that $g(u,\vec{v}) = \mathbf{1}$ and at least there is one u for which M never halts, N does not halt. □

Remark 6.39. Note that for any decidable relation $R(u,\vec{v})$, the function $f(\vec{v}) = \mu|u|.R(u,\vec{v})$ computing $\mathbf{1}^n$ where n is the length of the shortest possible u such that $R(u,\vec{v})$. The reason is that $R(u,\vec{v})$ is nothing but $\chi_R(u,\vec{v}) = \mathbf{1}$.

Example 6.40. The empty function $empty(w)$ with the empty domain is computable because $empty(w) = \mu|u|.\ [Z(u) = \mathbf{1}]$.

Example 6.41. Let $R \subseteq \mathbb{N}$ be a c.e. relation and $f : \mathbb{N} \to \mathbb{N}$ be a computable function over the alphabets $\{1\}$. Then $f|_R$, the restriction of f to the relation R is also computable because $f|_R(v)$ is definable by $f(\mu|w|.\ [R(w) \wedge (w = v)])$.

Example 6.42. Let $R \subseteq \Sigma^*$ be a decidable relation. Then it is also c.e. because $\chi'_R(v)$ is definable by $C_{\mathbf{1}}(\mu|w|.\ [R(w) \wedge (w = v)])$.

Example 6.43. The function $Twin(n)$ that computes the least $p \geq n$ such that both p and $p + 2$ are primes is computable because it is definable via unbounded search $Twin(n) = \mu|m|.[Prime(m) \wedge Prime(m + 2) \wedge n \leq m]$. Note that the totality of the function $Twin$ is equivalent to the twin prime conjecture stating the existence of infinitely many pairs of primes $(p, p + 2)$.

Theorem 6.44. *(Kleene)*

(*i*) *A function is computable iff it is constructible from the basic functions, composition, primitive recursion and unbounded search.*

(*ii*) *A relation is c.e. iff it is in the form* $\exists u f(u, \vec{v}) = \mathbf{1}$ *where* f *is primitive recursive.*

Over the language $\{1\}^*$, it is also possible to prove a far more powerful characterization of c.e. relations using polynomials instead of arbitrary primitive recursive functions. This characterization was proved by Yuri Matiyasevich, Julia Robinson, Martin Davis and Hilary Putnam, hence its acronym:

Theorem 6.45. *(MRDP) Over the language* $\{1\}^*$*, a relation is c.e. iff it is in the form* $\exists \vec{y} p(\vec{x}, \vec{y}) = 0$ *where* p *is a polynomial.*

So far we have provided a machine-independent characterization of computable functions and computably enumerable relations. In the following theorem, we reduce the notion of decidability to the notion of computably enumerability to provide a characterization for decidable relations, as well:

Theorem 6.46. *A relation* R *is decidable iff both* R *and* R^c *are c.e. Therefore,* R *is decidable if there are primitive recursive functions* f, g *such that* $R(\vec{v})$ *iff* $\exists u f(u, \vec{v}) = \mathbf{1}$ *iff* $\forall w g(w, \vec{v}) = \mathbf{1}$.

Proof. It is enough to prove the first part. The rest follows. For the first part, one direction is proved in the Example 4.25. For the other direction, if both R and R^c are c.e., they have algorithms M and N. For a decision algorithm K for R, run M and N in parallel. More formally, use a 2-taped

Turing machine and copy the input on the second tape, as well. Then run one step of M on the first tape and one step of N on the second tape, alternately, till one of them halts. Since for any $\vec{v}$, either $\vec{v} \in R$ or $\vec{v} \in R^c$. Therefore, one and exactly one of M and N halts and outputs **1**. If M halts, output **1**, if N halts, output **0**. □

7 The Stability Theorem

Turing machines are defined in a way that depends on the given set of alphabets and computability inherits this unintended dependence from them. However, as the reader may expect, we want the notion of comutability independent of all these details. For instance, this would be unsatisfactory if we have a numeral function computable over the binary expansions of the numbers but uncomputable over the usual unary representation. The problem, though, is with the functions themselves. A function is defined over a fixed set of alphabets and it is somewhat meaningless to compare functions with different possible inputs and outputs. To solve this problem, we need a transfering method to transfer functions over one set of alphabets to the functions over another set, respecting computability in boh ways. This section is devoted to this kind of *stability*.

Let Σ be a set of alphabets with at least two elements $a, b \in \Sigma$.[3] And let $x \notin \Sigma$ be a new symbol. We want to present a translation from the expressions over the alphabet $\Gamma = \Sigma \cup \{x\}$ into the expressions over the set Σ, in a way that the computable functions over Γ make a one to one correspondence with the computable functions over Σ. For this matter we need two translation functions; one for the encoding process $e : \Gamma^* \to \Sigma^*$ and the other for the decoding process $d : \Sigma^* \to \Gamma^*$. For the encoding function $e : \Gamma^* \to \Sigma^*$, define $e(w_1 w_2 \dots w_n)$ as $u_1 u_2 \dots u_n$ where u_i is defined in the following manner: If $w_i \in \Sigma$, then $u_i = w_i w_i$. If $w_i = x$, then u_i is ab. For instance, if $\Sigma = \{a, b\}$, then $e(axb) = aaabbb$. Note that the function e is total and computable if we consider it as a function over Γ. It simply reads the input and make any letter double. Moreover, if we restrict e to Σ^*, it will be again a very easy computable function. This restricted function is definable over Σ and we denote it by $e' : \Sigma^* \to \Sigma^*$. For the decoding process $d : \Sigma^* \to \Gamma^*$, read a string over Σ. If the length is odd, then just go right forever, and does not halt. Otherwise, Split the input into the blocks with length two and compute

[3]This is not a real restriction. All the results of this section also hold for a singleton Σ. But, the needed techniques must change dramatically and we want to avoid such complicacies in this very short lecture.

the inverse of the process that we defined above for each block. If all the blocks behave in an expected manner you can recover the original string over Γ. Otherwise, go right forever and does not halt. Note that d is not a total function. It works only on the image of the function e and is computable if we consider it as a function over Γ. Moreover, note that e and d are inverses, i.e., for any $w \in \Gamma^*$ we have $d(e(w)) = w$ and for any $u \in \Sigma^*$ for which d is defined, $de(u) = u$. Note that restricting d to strings without ab lands in Σ^* and is computable. This function that is clearly the inverse of the function e' is denoted by $d' : \Sigma^* \to \Sigma^*$.

Note that the whole processes of e and d are intuitively easy, syntactical and "computable". However, they can not be computable in its technical sense, because the notion of computability is defined over one fixed language and we do not have computability of functions between strings over different alphabets. However, it is possible to formalize this informal computability of e and d via the bridge role they play in transferring computable functions over Γ to computable functions over Σ and vice verse:

Theorem 7.1. *(Stability Theorem) Let $\Gamma = \Sigma \cup \{x\}$ where $x \notin \Sigma$. Then a function $f : (\Gamma^*)^k \to \Gamma^*$ is computable iff the function $\tilde{f} : (\Sigma^*)^k \to \Sigma^*$ by the definition $\tilde{f}(\vec{w}) = e(f(d(\vec{w})))$ is computable.*

Proof. Let us assume that $\tilde{f} : (\Sigma^*)^k \to \Sigma^*$ is computable. Then we show that f is also computable. Since $\tilde{f}$ is computable, thre exists a Turing machine M over Σ that computes $\tilde{f}$. First, we want to change this machine to a machine over Γ. The only thing to do is defining a dummy work for the machine when it reads x. We require the machine to state in the same state with the same head position. Then clearly, this new machine ignores any letter x in the input and does what $\tilde{f}$ demands. Now, combine this new machine with the machines for e and d when we consider them as computable functions over Γ. For the converse, assume that f is computable via a machine N over Γ and we want to show $\tilde{f}$ is computable over Σ. We will define the machine M simulating N in the following way: Read and write everything in pairs. If you read ww for $w \in \Sigma$, then do what N does for w, and if it writes $u \in \Sigma$, write uu, if you read ab, then do what N does for x and if it writes x, write ab. In any case, if you can not follow this pattern, go right forever. This is easily possible by making all the states of N double. Then this new machine computes $\tilde{f}$. □

Remark 7.2. Using the stability theorem, it is possible to transfer any computable function over Γ to a computable function over Σ. Sometimes, we use this theorem loosely to forget which language we are computing over.

For instance, note that adding one new symbol to the alphabets makes the language powerful enough to talk about all sequences of strings over the original alphabet. It is enough to add the symbol ";" to Σ to represent a sequence $w_1, w_2, \ldots w_n$ where $w_i \in \Sigma^*$ by one string $w_1; w_2; \ldots; w_n$ over $\Sigma \cup \{;\}$. In this case, sometimes, we silently move to a bigger alphabet to have the power to encode sequences. But we actually mean going to the bigger language, doing the computation and then using the stability theorem to come back.

Remark 7.3. Note that the map $e' = e|_{\Sigma^*}$ translates the set $\Sigma^* \subseteq \Gamma^*$ into the set A, consisting of all strings in the form $w_1 w_1 w_2 w_2 \ldots w_n w_n$ where $w_i \in \Sigma$ and $d' = d|_A$ acts as the inverse of e'. Moreover, we know that both of the functions e' and d' are computable over Σ. Therefore, in transfering the functions over Γ to the functions over Σ, if the domain or the range of any variable is $\Sigma^* \subseteq \Gamma^*$, we can use the computable equivalence of A and Σ^* to keep those variables unchanged. For instance, if $f : \Sigma^* \to \Sigma^*$ is computable over Γ, it implies that $\tilde{f}$ is computable over Σ^*. Define $g : \Sigma^* \to \Sigma^*$ as $d'\tilde{f}e'$. Since e' and d' are computable over Σ, then g is also computable over Σ. But $g(w) = d'e\tilde{f}d(e'(w))$. Since $de' = d'e = id$, we have $g(w) = f(w)$. Hence, f is also computable over Σ, meaning that in transfering the function we could keep the variable $w \in \Sigma^*$ and the range $f(w) \in \Sigma^*$ unchanged. Moreover, it shows that in computing a function over Σ, it is possible to extend the language temporarily for conveninece in the computation. It does not change the status of computability of the function.

Remark 7.4. Note that the stability theorem provides a translation pair for any pair of alphabet sets. The reason is that using the stability theorem we know how to add or eliminate an element from the alphabets. With these two operations at hand, we can clearly reach any alphabet set from any other one.

8 The Universal Machine

Any computable task can be implemented by a Turing machine and hence it is mechanizable. However, this mechanization is not necessarily in a uniform manner and for different tasks, we need to develop different machines; one machine for addition, one for multiplication, one for exponentiation and so on. To use the everyday life examples, it is similar to the pre-digital era of the single task machines such as telephone, the typing machine, Television and so on. What Alan Turing's gmae-changing insight added to that story was the existence of one multi-task machine capable of implementing every

possible computable task. This machine is called the universal machine and it can be rightfully called the heart of the modern digital age.

A universal machine is a Turing machine that reads a program (a Turing machine itself) and its input and simulate the work of that machine on that input. In this sense we have one machine to do every possible computation and to mechanize every computable task, it is enough to design one universal machine, once and for all. To have a machine like this, we have to first feed the machine with a program which means that our first task in this section is encoding the Turing machines into the strings of the language itself. Our strategy is the following. We first extend the alphabets from the original set Σ to the enriched alphabets $\bar{\Sigma} = \Sigma \cup \{1, \mapsto, ;, +, -, L, R, (,), [,], \langle, \rangle\}$ to have the power to freely talk about the programs and computations. Then we will use the Stability theorem to transfer whatever we have constructed over this new language to the original language. Therefore, throughout this section, we work in the extended language carelessly. Now let us formalize all the primitive notions of computations via this new language. There are three fundamental notions that we want to handle; algorithms, configurations and computations:

- Any Turing machine is nothing but a finite tableaux consisting of its basic data and its basic rules. Therefore, it is reasonable to think of the possibility of encoding these machines by some strings over a suitable set of alphabets. The following is one way to do that which is by no means canonical. Let M be a Turing machine. Represent M by the string $(I_1; I_2; \ldots; I_n)$ over $\bar{\Sigma}^*$ where I_i's are strings in the form $1^r; a \mapsto 1^s; b; D$ where $r, s \in \mathbb{N}$, $a, b \in \Sigma$ and $D \in \{L, R\}$, encoding the rule that if the machine in the state q_r reads a, it changes the content of the cell to b, goes to the state q_s and moves its head to left or right according to D.

- By the configuration of a machine at the moment n, we mean all the data of the machine at the n-th instance of time, namely, the tuple consisting of the state of the machine, its head position and the whole non-blank data over its tape, at the moment n. The configuration at 0 is called the starting configuration and a configuration whose state is in the halting states is called a halting configuration. A configuration is also a finite data and hence representable. For instance, we can represent a configuration by $[1^r; 1^s; x; E_1; E_2; \ldots, E_m]$ where E_i is a string in the form $(1^j; y; a)$ where $r, s, j \in \mathbb{N}$, $x, y \in \{+, -\}$ and $a \in \Sigma$, encoding that the state is q_r, the head is in the position $+s$ or $-s$ depending on

x and the content of the $(+j)$-th cell or $(-j)$-th cell is a. If j does not occur in E_i's, we assume that the content of that cell is blank.

- A computation is a sequence of configurations of the machine, starting with the initial configuration and following the rules of the machine. Represent a computation by $\langle C_1; \ldots; C_m \rangle$ where any C_i is a representation of the configuration of the machine in the i-th step.

Theorem 8.1. *(i) The relation $T(m, \vec{u}, w, v)$ that states "the machine m on the input $\vec{u}$ has the halting computation w leading to the output v" over the alphabet $\bar{\Sigma}$ is decidable.*

(ii) There exists a universal Turing machine $U(m, \vec{u})$ simulating all Turing machines, i.e., for any machine M with description m and any input $\vec{u}$ we have $U(m, \vec{u}) = M(\vec{u})$.

Proof. For (i), define the function $F(x, m, \vec{u})$ as a function over $\bar{\Sigma}$ that sends $(x, m, \vec{u})$ to the computation of m on $\vec{u}$ after $|x|$ many steps. This function is computable via the following natural `Simulation` algorithm that uses recursion on x. For the base case, `Simulation` generates the initial configuration by printing the initial state, the input itself and the head position over the first letter of the input. Then reading any element of x, first the machine finds the last configuration of m encoded in the output of the last step and then goes to m to see what the rules in m says about changing this last configuration. It then applies what m states to compute the next configuration and finally adds this new configuration to the whole string of computation that it has produced so far. The machine halts when it reads all the elements of x. Now, computing F, we can define $T'(m, \vec{u}, w)$ stating that "the machine m halts on the input $\vec{u}$ with the computation w" as

$$[(\mu|x| \leq |w|.[F(x, m, \vec{u}) = w]) \neq \mathbf{1}^{|w|+1}] \wedge hState(m, w)$$

where $hState(m, w)$ means that the last state of the computation w is halting according to m. Note that $hState(m, w)$ is computable because for its decision it is enough to scan both m and w. Then define the predicate $T(m, \vec{u}, w, v)$ as $T'(m, \vec{u}, w) \wedge (out(w) = v)$ where $out(w)$ computes the output of the computation w. Again note that $out(w)$ is computable because it is enough to scan the computation w to extract the output of the computation m from it as the tape data of the last configuration. For (ii), define $U(m, \vec{u}) = out(F(x_{(m,\vec{u})}, m, \vec{u}))$ where $x_{(m,\vec{u})} = \mu|x|.\ hState(m, F(x, m, \vec{u}))$. □

Remark 8.2. Using the Stability under the language extensions, both of the predicate T and the universal machine U can be transferred over Σ via the suitable encoding-decoding process.

The previous theorem can be rewritten more carefully to serve also as a proof for the Kleene's machine-independent characterization theorem:

Proof. For (i), one direction is already proved. For the other direction, note that a more detailed investigation shows that what we explained before is actually a primitive recursive construction for the predicate T. Therefore, the universal machine is constructible via Kleene's operations because it applies one unbounded search on the predicate T. Therefore, if a function $f(\vec{u})$ is computable via a Turing machine M with the code m, then $f(\vec{u}) = U(m, \vec{u})$. Since U is constructible, f is also constructible. For (ii), again one direction is proved. For the other direction, assume that the machine for R is M with the code m. Then note that $R(\vec{u})$ if $\exists w\ T(m, \vec{u}, w, \mathbf{1})$. This completes the proof. Note that this proof works over the extended language $\bar{\Sigma}$. To move it over the original language Σ, we also need a primitive recursive version of the translations in the Stability theorem. This is also possible but it needs more work. □

On Computably Enumerable Relations

Using what we have developed in this section we can explain the terminology that we use for c.e. relations:

Theorem 8.3. *A set $A \subseteq \Sigma^*$ is c.e. iff it is either empty or the range of a total computable function $f : \{\mathbf{1}\}^* \to \Sigma^*$.*

Proof. For simplicity, identify $\{\mathbf{1}\}^*$ with $\mathbb{N}$ and represent $\mathbf{1}^n$ by n. Then assume $A \neq \emptyset$ and M is the machine for A with the code m. Pick $a \in A$ and consider the predicate $R(u, w) = T(m, u, w, \mathbf{1})$ representing that "w is a computation of the machine M on the input u that halts and outputs $\mathbf{1}$". Let $en : \Sigma^* \to \Sigma^*$ be a total computable function whose restriction to $\{\mathbf{1}\}^*$ is surjective. Now, define $f : \{\mathbf{1}\}^* \to \Sigma^*$ as

$$f(n) = \begin{cases} u_n & R(u_n, v_n) \\ a & \neg R(u_n, v_n) \end{cases}$$

where $en(n_0) = u_n$ and $en(n_1) = v_n$ where $n + 1 = 2^{n_0}(2n_1 + 1)$. Firstly the range of f is clearly a subset of A because in the first case, v_n is the computation of the machine M on u_n with answer $\mathbf{1}$, meaning that $u_n \in A$. The second case is clear because $a \in A$. Conversely, if $b \in A$, then M halts on b and outputs $\mathbf{1}$. Take w as the computation of M on b. Then since the restriction of en to $\{\mathbf{1}\}^*$ is surjective, there exist m, k such that $en(m) = b$ and $en(k) = w$. Therefore, $f(n) = b$ where $n + 1 = 2^m(2k + 1)$. □

Remark 8.4. The Theorem 8.3 states that a set A is c.e. if it is either empty or there exists a total computable function with range A, surjective even when it is restricted to $\{\mathbf{1}\}^*$. In other words, f reads a number n and produces an element of A. Since $f|_{\{\mathbf{1}\}^*}$ is surjective, it covers A (with possible repetitions). This means that $A = \{f(0), f(1), f(2), \ldots\}$ meaning that f enumerates all the elements of A in a computable manner; hence the name computably enumerable.

9 The Uncomputable World

In this section we will present some negative results including an uncomputable function, an undecidable relation and a relation that is c.e. but not decidable. Our method is essentially different than what we did before in the pumping lemma for the regular languages. Here, the computability notion is extremely powerful. Therefore, it seems impossible to prove an undecidability result by investigating all the possible patterns of computation. However, this full notion of computation is so powerful that it can speak about itself, as we observed in the previous section. This power then leads to the usual self-referential paradoxes, this time encoded via computations. This self-referential type of argument is the main technique of proving undecidabilities.

Theorem 9.1. *The halting relation $H \subseteq \Sigma^* \times \Sigma^*$ defined as $H(m,u) = \exists w v T(m,u,w,v)$ is c.e. but not decidable. Hence, the total function χ_H is not computable.*

Proof. It is c.e. because it has the description $\exists w v T(m,u,w,v)$ where the predicate $T(m,u,w,v)$ is the decidable relation that states "w is the computation of the machine m on the input u halting and outputting v". For undecidability, let us assume that there exists a Turing machine M to decide H. Therefore, M always halts. Define the machine $N(x)$ as the machine that first compute $M(x,x)$ and then if it is one, it does not halt and if it is zero, it halts and outputs zero. Now, let us check $N(n)$ where n is the code of N. If N halts on n, then by definition of N, we have $M(n,n) = 0$ which means that N does not halt on n. If N does not halt on n, then again by definition $M(n,n) = 1$ which means N halts on n. □

Corollary 9.2. *The set H^c is not computably enumerable.*

Proof. If it is computably enumerable, then since H is also computably enumerable, by Theorem 6.46, H will be decidable which we know it is not. □

Remark 9.3. Note that the previous corrolary shows that the class of c.e. relations is not closed under complement.

In the following, we will generalize what we had for the halting property of a machine to a very general setting. We will show that any non-trivial property of the machines that depends on the function that the machine computes and not the implementation details (like its number of states) is not decidable.

Definition 9.4. Let C be a class of Turing machine. It is called *extensional* if for any computable function f, either C has all the machines for f or non of them.

Theorem 9.5. *(Rice Theorem) Let C be an extensional class of Turing machines. Then C is either undecidable or trivial, i.e., either empty or the whole set of all Turing machines.*

Proof. Assume C is decidable and non-trivial. Since C is extensional, either C has all the machines computing the function with empty domain or C has non of them. W.l.o.g. we assume the latter because in the first case we can use C^c in the rest of the argument. Now we use the decidability of C to show that halting relation H is also decidable. Since C is not trivial, there is at least one $K \in C$. The decision algorithm for halting is the following: Read a machine M and an input w. Then construct the machine $[M, w]$ with the input u and the following algorithm: Run M on w. If it halts, run K on u. It is clear that $[M, w]$ computes the same function as K does if M halts on w. Otherwise, the domain of $[M, w]$ is empty. Since C has non of the machines computing the function with the empty domain and $K \in C$, we have $[M, w] \in C$ iff M halts on w. Finally, we can run the decision procedure for C on $[M, w]$. □

Example 9.6. The class of all Turing machines halting on all of their inputs is not decidable because it is extensional and non-trivial.

Exercise 9.7. Show that the class of all Turing machines computing the constant functions is not decidable.

Finally, it is satisfying to end this section by the answer to Hilbert's tenth problem. It heavily depends on the elegant MRDP theorem that we stated before:

Theorem 9.8. *(Unsolvability of Hilbert's Tenth Problem) There is no algorithm to decide whether a polynomial with integer coefficients has an integer root or not.*

Proof. By MRDP, the halting relation $H(x, y)$ is equivalent to a statement like $\exists \vec{z} \in \mathbb{N}\ p(x, y, \vec{z}) = 0$. Define $q(x, y, \vec{z}, \vec{w})$ as

$$p(x, y, \vec{z})^2 + \Sigma_i (z_i - w_{i1}^2 - w_{i2}^2 - w_{i3}^2 - w_{i4}^2)^2$$

Then $\exists \vec{z} \in \mathbb{N}\ p(x, y, \vec{z}) = 0$ iff $\exists \vec{z}\vec{w} \in \mathbb{Z}\ q(x, y, \vec{z}, \vec{w}) = 0$. The main reason behind this equivalence is the Lagrange theorem stating that any natural number is representable as the sum of four perfect squares. Finally, if there exists an algorithm to decide the latter, then it means we can decide the former and hence the halting problem which is impossible. □

Finally, let us mention some other concrete undecidable problems. There are many of them scattered in different fields of mathematics, from topology and analysis to algebra and combinatoric. In the following we mention two problems of this kind:

Example 9.9. (The Matrix Mortality Problem) Use a reasonable language to talk about matrices with intefger entries, for instance $\{1, ; \}$. Then in this language, pick the set of all finite sets of $n \times n$ matrices with integer entries such that there exists a multiplication of them in some order, possibly with repetitions, to yield the zero matrix. This set is c.e. because it is definable via the unbounded existential quantifier (the list of multiplications) over a decidable relation "checking whether the multplication is zero". It is decidable because it only needs following the list, computing the multiplications and then checking wether it is zero or not. The set is undecidable.

Example 9.10. (Post Correspondence Problem) Fix Σ with at least two elements and use $\Sigma \cup \{1, ; \}$. Then pick the set of all pairs of finite lists of strings over Σ such as $\{a_i\}_{i=1}^n$ and $\{b_i\}_{i=1}^n$ with the same length such that there exists a list of indices like $\{i_j\}_{j=1}^m$ where $1 \leq i_j \leq n$ such that $a_{i_1} a_{i_2} \dots a_{i_m} = b_{i_1} b_{i_2} \dots b_{i_m}$. This problem is c.e. because it is definable via an unbounded existential quantifier (list of indices) over a decidable relation "over the given list $\{i_j\}_{j=1}^m$, we have $a_{i_1} a_{i_2} \dots a_{i_m} = b_{i_1} b_{i_2} \dots b_{i_m}$". This is decidable because it only needs following the list to put a_{i_j}'s and b_{i_j}'s and then checking whether they are equal by a simple scanning. The relation is undecidable.