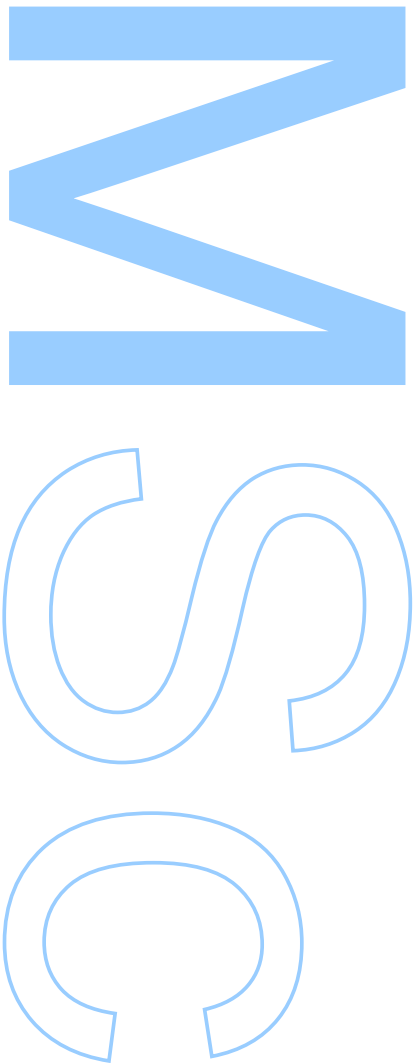
Kunz Languages

Jaume Usó i Cubertorer
Mestrado em Matemática
Departamento de Matemática
2023

Orientador
Prof. Dr. Manuel Delgado, Faculdade de Ciências




U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Sworn Statement

I, Jaume Usó i Cubertorer, enrolled in the Master Degree in Mathematics at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission. By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution. I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws. I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Jaume Usó i Cubertorer

June 28, 2023

Acknowledgements

Em primeiro lugar queria agradecer ao meu orientador, o Manuel Delgado em geral pelo seu trabalho e esforço, e em particular por ter conseguido encontrar um tema que me permitisse fazer o que lhe pedi: estudar linguagens formais.

Agraïsc a la meua família. Als meus pares per l'esforç que han fet, en especial a ma mare per insistir em que acabara esta etapa. I al meu germà, per vindre a vore'm i per estar sempre disponible per a comentar dubtes.

Agradeço também a todos os professores que tive no mestrado pelo seu trabalho que juntamente como os meus colegas fizeram que me sentisse muito bem na faculdade.

Muito obrigado Bernardo pela tua amizade e por tornar a minha estadia no Porto mais agradável e interessante.

Finalment, moltes gràcies a Albert, Carlos, Miguel, Pascual, Pau, Roger i Víctor per haver vingut a vore'm.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Matemática

Mestrado em Matemática

Linguagens de Kunz

por Jaume Usó i Cubertorer

Os semigrupos numéricos são subconjuntos dos inteiros não negativos que têm o zero como um dos seus elementos, são fechados para a adição e o seu complementar é finito. Devido ao facto de terem uma estrutura simples, aparecem no estudo de muitos problemas em diferentes áreas e é este facto o que desperta o nosso interesse neles e destaca a sua importância.

No campo dos semigrupos numéricos há questões com diferentes graus de dificuldade, isto é, algumas famílias de semigrupos numéricos parecem ser mais complexas do que outras. A motivação para esta tese é formalizar essa intuição ao dar uma escala de complexidade que tal vez possa ajudar a prever a dificuldade dos problemas.

Para realizar esta tarefa, associamos de forma única uma palavra a cada semigrupo numérico, e assim acabamos por associar uma linguagem a cada família de semigrupos numéricos. Depois, aplicamos teoria de linguagens formais para estudar e classificar essas linguagens, e em certo modo as famílias de semigrupos numéricos. Em particular vamos estudar as famílias formadas por todos os semigrupos numéricos duma certa profundidade e vemos que as linguagens associadas a familias de semigrupos numéricos de profundidade não superior a dois são menos complexas que aquelas linguagens que associamos a famílias de semigrupos de profundidade superior. As linguagens que aparecem de forma natural ao levar a cabo este estudo são o que chamamos de Linguagens de Kunz.

Palavras Chave: Semigrupos numéricos, linguagens de Kunz, hierarquia de Chomsky.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Matemática

MSc. Mathematics

Kunz Languages

by Jaume Usó i Cubertorer

Numerical semigroups are subsets of nonnegative integers that have zero as one of their elements, are closed under addition and have a finite complement. Due to the simplicity of this structure, numerical semigroups appear in the study of many problems in different areas and hence their importance and our interest in them.

Different problems in numerical semigroups have different degrees of difficulty, that is, some families of numerical semigroups seem to be more complex than others. The motivation for this thesis is to formalize this idea by giving a complexity scale that might help to predict the difficulty one can encounter in some problems.

For this task, we associate to each numerical semigroup a word in a unique way, and by doing this we are associating a language to each family of semigroups. Then, we apply language theory to study and classify these languages and somehow these families of numerical semigroups. In particular, we will study the families consisting of all numerical semigroups of a certain depth, and we will see that languages associated to families of numerical semigroups of depth not greater than two are less complex than those associated to semigroups with greater depth. The languages that naturally appear while performing this task are what we call Kunz languages.

Key Words: Numerical semigroups, Kunz languages, Chomsky hierarchy.

Contents

Acknowledgements iii

Resumo v

Abstract vii

Contents ix

List of Figures xi

Introduction 1

1 Numerical semigroups 3
1.1 Kunz Tuples 10

2 Formal Languages and Automata 13
2.1 Formal languages 13
2.1.1 Grammars and Automata 14
2.1.1.1 Grammars 14
2.1.1.2 Automata 18

3 Kunz Languages 33
3.1 Conclusions 51

A Details for lbas 53

List of Figures

1.1 Visual example of depth for $S = \langle 4, 5, 6 \rangle$ 9

2.1 Transition graph for M 19
2.2 Transition graph for M 21
2.3 Transition graph of an ndfa that recognizes $L = \{a^n b^n : n \in \mathbb{N}_0\} \cup \{a\}$ 26
2.4 Model of a Turing Machine. 27
2.5 Turing Machine that recognizes $L = \{a^n b^n c^n : n \in \mathbb{N}\}$ 29

3.1 dfa that recognizes K_0. 34
3.2 dfa that recognizes K_1. 35
3.3 dfa that recognizes K_2. 35
3.4 lba that accepts K_3. 42

A.1 lba that compares two natural numbers n and m. 54
A.2 Turing Machine that converts a natural number from base $n+1$ to unary notation. 55
A.3 lba that converts a natural number from unary to base $n+1$ notation. 56
A.4 Turing Machine that goes to index $i \in \{1, \ldots, |w|\}$. 56

Introduction

Numerical semigroups are subsets of nonnegative integers that have zero as one of their elements, are closed under addition and have a finite complement. Studying these structures usually involves investigating some notable elements and relations between them. Among the most important notable elements, there is the least positive element of the semigroup and the least element from which all integers belong to the semigroup. These are called, respectively, *multiplicity* and *conductor* and are usually denoted by m and c. An important relationship between these two elements is the *depth*, defined as $\left\lceil \frac{c}{m} \right\rceil$. In problems such as the *Wilf Conjecture* or *Counting numerical semigroups*, it appears that for higher values of depth, things turn out to be more difficult, more elaborated proofs are needed or even have not been found yet, see [2] and [6], respectively, for references.

Our goal in this thesis is to study this fact and constitute a possible formalization of the intuition that depth sets a complexity scale for numerical semigroups. That is, the family of numerical semigroups with a certain depth q is more complex than the family of semigroups with some depth lower than q. In future problems regarding families of numerical semigroups of some given depth, knowing this might help to anticipate the difficulty one can encounter.

To formalize our intuition, we will apply formal language theory to the study of numerical semigroups. Bringing these two branches of mathematics together relies on the fact that given a numerical semigroup of multiplicity m, there is a unique $(m-1)$-tuple associated to it, the Kunz tuple. Every numerical semigroup has its own Kunz tuple, there are no numerical semigroups sharing the same one. If instead of looking at a Kunz tuple as a tuple and we see it as a word, we can then injectively associate a word to every numerical semigroup. Taking advantage of the existence of such words, we can associate languages to families of semigroups of a certain depth, and define the *Kunz language* of depth q as the set of all words associated to numerical semigroups of depth q. With this, we convert the task of studying the complexity of the family of numerical semigroups with depth q to

the study of the complexity of the Kunz language of depth q. Once we have reached this point, we can use the *Chomsky hierarchy* to classify Kunz languages, and consequently families numerical semigroups. The work carried out in this thesis gave rise to the pre-print [3], which has been submitted for publication.

The thesis is structured in three chapters. The first one is dedicated to numerical semigroups, the second one to formal languages and the last one includes all the results of the classification of Kunz languages applying what has been mentioned in the previous chapters.

In Chapter 1 we start by giving a definition and some examples of numerical semigroups. Then we introduce some basic concepts and explain how and why the study of semigroups appeared. After that, we define some important elements of these structures and relations between them, paying special attention to the depth. Finally, we get to define Kunz tuples.

Chapter 2 deals with the necessary content regarding formal languages. Here we start by defining what a formal language is and giving some examples. There are essentially two ways to approach the study of languages: grammars and automata. We will begin with the former and introduce the Chomsky hierarchy, as it was originally done. Afterwards, we will continue with the latter and talk about automata and explain how for our purpose of defining types of languages they are equivalent to grammars.

Finally, Chapter 3 is the last and most important chapter, since it is the one containing all the results. The first thing we do here is to define Kunz languages, the object of study in this thesis. Afterwards, step by step we start to classify these languages by applying concepts from the previous chapters. While doing this classification we introduce some results that we need and whenever possible we try to connect what we do with other well-known results. Finally, we summarize some conclusions and draw some lines on how this work could be extended.

At the end of the thesis, we added Appendix A in which we devoted some time to give details of proofs developed throughout the third chapter.

Chapter 1

Numerical semigroups

In this first chapter, we will introduce the concept of numerical semigroup. Our goal is to give a proper definition as well as a context of how they historically appeared. Then, step by step, we will introduce some basic concepts and results until we can define the depth of a numerical semigroup. This concept of depth will open the gate to Kunz languages. For this chapter, we have mainly used [10] for the theory on numerical semigroups, [9] for questions related to the Frobenius problem and finally [15] for Kunz tuples. Let us now begin with the definition of numerical semigroup.

Definition 1.1. A numerical semigroup is a subset $S \subseteq \mathbb{N}_0$ of the non-negative integers such that:

- $0 \in S$.
- S is closed under addition.
- $|\mathbb{N}_0 \backslash S|$ is finite.

A trivial example of a numerical semigroup is the whole set $\mathbb{N}_0$, since 0 is one of its elements, it is closed under addition, and its complement is the empty set, which is finite. In order to see more examples of numerical semigroups, let us think about how to build them. Given a subset $A \subseteq \mathbb{N}$ of the set of all positive integers, we will denote by

$$\langle A \rangle = \{\lambda_1 a_1 + \cdots + \lambda_n a_n | \lambda_1, \ldots, \lambda_n \in \mathbb{N}_0, a_1, \ldots, a_n \in A\}$$

the set of all linear combinations over $\mathbb{N}_0$ of elements in A. It is clear that for any set A we will have that $\langle A \rangle$ is closed under addition and contains 0. Then it is natural to raise the question of when will it have a finite complement, that is, when will it be a numerical semigroup. In the following result, we can see that the answer is simple: when $\gcd(A) = 1$.

Lemma 1.2. *Let $A \subseteq \mathbb{N}$ be a nonempty set. Then $\langle A \rangle$ is a numerical semigroup if and only if* $\gcd(A) = 1$

Proof. As it has been said, we only need to worry about $\mathbb{N} \backslash \langle A \rangle$ being finite.
($\rightarrow$) Let $A \subseteq \mathbb{N}$ be such that $\langle A \rangle$ is a numerical semigroup and let $d = \gcd(A)$. If some $s \in S$, then $d|s$. Since $\langle A \rangle$ is a numerical semigroup, its complement has to be finite and as a consequence, there has to be an $x \in \mathbb{N}$ such that $d|x$ and $d|x+1$ and so we conclude that d has to be 1.
($\leftarrow$) Let $A \subseteq \mathbb{N}$ be such that $1 = \gcd(A)$. Due to generalized Bézout's identity, here exist $z_1, \dots, z_n \in \mathbb{Z}$ and $a_1, \dots, a_n \in A$ such that $z_1 a_1 + \cdots + z_n a_n = 1$. Now, if we move every negative term z_i to the right,we get an equality of positive terms. That is, let $i_1, \dots, i_k$ be the indices of positive z_i and $j_1, \dots, j_l$ the indices of the negative z_i then we have

$$\underbrace{z_{i_1} a_{i_1} + \cdots + z_{i_k} a_{i_k}}_{s+1} = 1 \underbrace{- z_{j_1} a_{j_1} - \cdots - z_{j_l} a_{j_l}}_{s}$$

So there exists $s \in \langle A \rangle$ such that $s+1 \in \langle A \rangle$. We will now prove that if $n \geq (s-1)s + (s-1)$ then $n \in \langle A \rangle$, what will mean that the complement of $\langle A \rangle$ is finite.
Let $n \geq (s-1)s + (s-1)$. By the division algorithm there exist q and r such that $n = qs + r$ with $0 \leq r < s$. Then $n = rs + r + qs - rs = (rs + r) + (q - r)s = r(s+1) + (q-r)s$. Since $n \geq (s-1)s + (s-1)$ we have that $q \geq s - 1 \geq r$, so $q - r \geq 0$ and then $n \in \langle A \rangle$ as it is a linear combination with non-negative integer coefficients of s and $s+1$. □

With this, we can now easily think of a numerical semigroup by giving a set of positive integers A with $\gcd(A) = 1$ that will generate it. Let's see now some non-trivial examples of numerical semigroups.

Example 1.1.
- $\langle 2, 3 \rangle = \{0, 2, 3, 4, \rightarrow\}$.
- $\langle 3, 5, 7 \rangle = \{0, 3, 5, 6, 7, 8, \rightarrow\}$.

The arrow $\rightarrow$ means that every number greater than the number preceding the arrow belongs to the semigroup.
Let us also see an example to show that it is indeed important that $\gcd(A) = 1$ otherwise we have an infinite number of gaps.

- $\langle 2, 4 \rangle = \{0, 2, 4, 6, 8, \dots\} = 2\mathbb{N}_0$ which leaves all odd numbers in the complement thus being not a numerical semigroup.

The way of constructing numerical semigroups given a finite number of coprime generators is very convenient. Furthermore, given a numerical semigroup S there always exist $s_1, s_2, \dots, s_n \in \mathbb{N}$ such that $\langle s_1, s_2, \dots, s_n \rangle = S$. In other words, every numerical semigroup has a finite set of generators, and moreover, there is a unique minimal set of generators under inclusion. The elements of this minimal set of generators are usually referred to as the *minimal generators* of the semigroup. The existence of the minimal generating set is not at all obvious, we can see how it is indeed true in the following theorem.

Theorem 1.3. *Every numerical semigroup admits a unique minimal set of generators under inclusion. Moreover, this minimal set of generators is finite.*

Proof. Given a numerical semigroup S we will write $S^* = S \backslash \{0\}$ to refer to the positive elements of S.
We will prove that the set $A = S^* \backslash (S^* + S^*)$ is the minimal set of generators of S and that it is finite.
First, let us see that A is a system of generators. Let $s \in S^*$, if $s \notin A$ then there exist $x, y \in S^*$ such that $s = x + y$. Now, either both $x, y \in A$ or at least one of them does not, $x \notin A$ (or $y \notin A$). In the second case, we can repeat the previous process, and after a finite number of steps, we will have found $s_1, s_2, \dots, s_n \in A$ such that $s = s_1 + s_2 + \cdots + s_n$.
Now, we will prove the minimality of A. Let B be a system of generators of S and $a \in A$, then there have to be $n, \lambda_1, \dots, \lambda_n \in \mathbb{N}$ and $b_1, b_2, \dots, b_n \in B$ such that $a = \lambda_1 b_1 + \cdots + \lambda_n b_n$. Since $a \notin (S^* + S^*)$ we deduce that $a = b_i$ for some $i \in \{1, 2, \dots, n\}$ and we can conclude that $a \in B$ for any $a \in A$, that is, $A \subseteq B$.
Finally, we have to see that the set A is indeed finite. Let m be the smallest positive number of S and $c \in S$ be such that $(c + i) \in S$ for any $i \in \mathbb{N}$. Then for every $n \in \mathbb{N}$ with $n \geq (m + c)$ we have that $n \in (S^* + S^*)$ and consequently $|A| = |S^* \backslash (S^* + S^*)| \leq (m + c)$ and thus A is finite. □

Remark 1.4. It is important to remark that even though numerical semigroups are *semigroups* they are actually monoids, since along with being closed under addition, they have a neutral element. In fact, they are submonoids of the monoid $\mathbb{N}_0$ and some authors refer to numerical semigroups as numerical monoids.

While proving Theorem 1.3 we have referred to some special elements of a semigroup. Namely the smallest positive element, which we denoted by m, and the smallest positive number such that any number greater than it is contained in the semigroup, which we

called c. These are two important invariants of numerical semigroups: respectively the *multiplicity* and the *conductor*. The conductor though is usually defined using the Frobenius number, which is the biggest integer that does not belong to the semigroup. Then the conductor is the Frobenius number plus one.

It is worth now devoting some time to explain where the Frobenius number comes from since it can be in some way the origin of the study of numerical semigroups. Afterwards, we will see some other invariants.

The Frobenius Problem

The traditional problem, called the coin problem, is the following: given coins of different monetary values (natural values), find the largest amount that cannot be paid using those coins.

A similar problem, referred to as the Frobenius problem, is known to be proposed by Frobenius in his lectures but already with some more mathematical structure. He asked for a formula for the largest integer that is not representable as a linear combination over non-negative integers of a set of natural numbers whose greatest common divisor is one. As we can see, this is equivalent to finding a formula for the solution of the coin problem with the addition of the greatest common divisor, which guarantees the existence of a solution. Sylvester introduced a related problem, sometimes known as the Sylvester problem, that was to determine how many natural numbers do not have such representation.

It is not hard to see that the first problem is that of given $\emptyset \neq A \subseteq \mathbb{N}$ with $\gcd(A) = 1$ find a formula in terms of the elements of A of the so-called Frobenius number. In other words, find a formula for the Frobenius number of a numerical semigroup in terms of its generators.

The Sylvester problem, on the other hand, would be to find out the number of gaps of the given numerical semigroup S, that is, to find $|\mathbb{N}_0 \backslash S|$.

These problems that drew the attention of Frobenius and Sylvester towards the end of the 19^{th} century are seen as the beginnings of numerical semigroups.

Many details about the Frobenius Problem, such as algorithms for its computation, related problems and applications, can be consulted in [9].

Let us now summarize and give proper detailed definitions and notation for the previous concepts and introduce some more. Moreover, to exemplify these concepts, we will show

what are the invariants over the numerical semigroups $S_1 = \langle 2, 3\rangle$ and $S_2 = \langle 3, 5, 7\rangle$, those of Example 1.1.

Definition 1.5. Given a numerical semigroup S the *Frobenius number* of s $F(S)$ is the greatest integer that does not belong to S.
$F(S_1) = 1$ and $F(S_2) = 4$.

Definition 1.6. The *conductor* $c(S)$ of a numerical semigroup S is defined as $c(S) = F(S) + 1$.
$c(S_1) = 2$ and $c(S_2) = 5$.

Definition 1.7. The *multiplicity* $m(S)$ of a numerical semigroup S is the least positive number that belongs to S.
$m(S_1) = 2$ and $m(S_2) = 3$.

Definition 1.8. Given a numerical semigroup S whose minimal system of generators is $\{s_1, s_2, \dots, s_n\}$ we call its cardinality n, the *embedding dimension* of S, and we denote it by $e(S)$.
$e(S_1) = 2$ and $e(S_2) = 3$.

Definition 1.9. Let S be a numerical semigroup, the *genus* of S, denoted $g(S)$, is $|\mathbb{N}_0 \backslash S|$. The genus is sometimes referred to as the degree of singularity of S.
$g(S_1) = 1$ and $g(S_2) = 2$.

Continuing with the Sylvester problem, with our terminology, it would be the problem of finding the genus of the semigroup. It is interesting how the embedding dimension plays a role in both problems. As for now, for embedding dimension greater than two there is no known formula for the Frobenius number nor the genus. It is known though, that it cannot exist a polynomial formula for the Frobenius number in those cases. For embedding dimensions one and two, the situation is considerably simpler as we can see next.

For embedding dimension one there is only one numerical semigroup, the trivial one $N_0 = \langle 1\rangle$, and so we have that $F(S) = -1$ and $g(S) = 0$. Let us now look at the case of embedding dimension two, which is quite more satisfying. To begin with, there is an infinite number of numerical semigroups with this condition.

Proposition 1.10. *Let* $a, b \in \mathbb{N}$ *such that* $\gcd(a, b) = 1$ *then, for the numerical semigroup* $S = \langle a, b\rangle$ *we have that:*

- $F(S) = ab - a - b$,
- $g(S) = \dfrac{ab - a - b + 1}{2}$.

Proof. First, we will start by proving the statement about the Frobenius number. For the sake of contradiction suppose that $ab - a - b$ is representable as $\alpha a + \beta b$ for some $\alpha, \beta \in \mathbb{N}_0$. Then, $(ab - a - b) \equiv (\alpha a + \beta b) \pmod{a}$, and so we have $-b \equiv \beta b \pmod{a}$, and then $\beta \equiv -1 \pmod{a}$. In the same way, we get to $\alpha \equiv -1 \pmod{b}$. But with this we have that $ab - a - b = \alpha a + \beta b \geq (b-1)a + (a-1)b = 2ab - b - a$ which is a contradiction, so $(ab - a - b) \notin S$. Now we need to prove that for any $s \in \mathbb{N}_0$ such that $s > (ab - a - b)$ it is true that $s \in S$. By the extended Euclidean algorithm, we know that there exists $\lambda, \mu \in \mathbb{Z}$ such that $\lambda a + \mu b = n$ for all $n \in \mathbb{N}_0$ since for any $k \in \mathbb{N}$ we have that $(\lambda + kb)a + (\mu - ka)b = n$ we can assume $\lambda > 0$. Among all possible positive λ verifying $\lambda a + \mu b = n$, we choose the smallest one. Notice that this λ is such that $0 \leq \lambda \leq b - 1$, otherwise the choice $\lambda' = (\lambda - b)$ would be a smaller λ contradicting our choice.

Now, if $s \geq (a-1)(b-1) = ab - a - b + 1$, since $0 \leq \lambda \leq b - 1$ we have that $\mu b = s - \lambda a \geq (a-1)(b-1) - (b-1)a = 1 - b > -b$ and consequently $b(\mu + 1) > 0$ so $\mu \geq 0$ as we wanted to prove.

Now, studying the case of the genus, what we will do is prove that for numbers less than or equal to $F(S)$ there is a bijection between those numbers in S and those that do not belong to it.

So, let $\mathbb{N}_{\leq F} = \{n \in \mathbb{N}_0 \mid n \leq F(S)\}$, and let $A = \mathbb{N}_{\leq F} \cap S$ and $B = \mathbb{N}_{\leq F} \cap (\mathbb{N}_0 \backslash S)$, we can define

$$\begin{array}{rccl} \varphi: & A & \longrightarrow & B; \\ & s & \mapsto & F(S) - s. \end{array}$$

if $s \in S$, it is clear that $(F(S) - s) \notin S$, otherwise $F(S) = s + (F(s) - s)$ would be in S which is a contradiction. Moreover, if $s \in A$ we have that $0 < F(s) - s \leq F(S)$, and hence $(F(S) - s) \in B$. So we have that φ is well defined, and it is clear that it is injective. Next, we want to see that it is surjective as well. In fact we will see that φ is its own inverse. Let $h \in B$, we want to see that $(F(S) - h) \in A$. As $\gcd(a, b) = 1$ there exist integers x, y such that $h = xa + yb$, and one of them has to be negative. Otherwise it would contradict the fact that $h \notin S$. Let us assume without loss of generality that y is negative. So we can write $h = xa - yb$ for $x, y > 0$. In addition, as for any $k \in \mathbb{N}_0$ it is true that $h = (x - kb)a - (y - ka)b$ we may also assume that $x < b$. Note that for a and b, one of the coefficients has to be positive and the other negative. So $(y - ka)$ will remain positive as long as $(x - kb)$ is positive as well. Now, from this and from the fact that $F(S) = ab - a - b$, we have

$F(S) - h = ab - a - b - ax + by$. And then $F(S) - h = \underbrace{(b - x - 1)}_{\geq 0}a + \underbrace{(-y - 1)}_{\geq 0}b \in S$. That is, we have that φ is surjective and consequently bijective.

So $|A| = |B|$, and as $|A| + |B| = |\mathbb{N}_F| = ab - a - b + 1$ we have that the number of gaps is $g(S) = |B| = \dfrac{ab - a - b + 1}{2}$. □

So we have seen how the embedding dimension sets a complexity scale for numerical semigroups in terms of computation. For embedding dimension 1, the case is trivial, for embedding dimension 2 a simple and nice formula does the trick to compute $F(S)$, but for greater values of e, we do not know of the existence of a formula but we do know that it will not be as simple as a polynomial. What we aim to do in this thesis is to see if something similar happens with the depth of a semigroup. To begin with, let us see what depth is and why does it attract our interest.

Definition 1.11. The depth $q(S)$ of a numerical semigroup S is defined as $q(S) := \left\lceil \dfrac{c(S)}{m(S)} \right\rceil$.

Example 1.2.

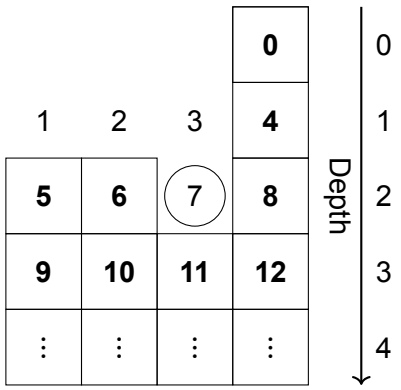


Figure 1.1: Visual example of depth for $S = \langle 4, 5, 6 \rangle$

The idea behind why the depth of a semigroup S has that name, is that if we order all non-negative numbers in rows of length the multiplicity m, and paint in a different colour or draw a box around the numbers belonging to S we will end up with m columns. Each one of these columns will have its own height, leaving some gaps on top of them. The size of the deepest of these gaps is the depth q of the semigroup. Note that it will be located in the column containing the Frobenius number F. In this example we took $S = \langle 4, 5, 6 \rangle$ which has depth $q = 2$ and Frobenius number 7 which is circled.

Even though the concept of depth was first introduced in [4], it had already appeared before but without receiving a name. An interesting fact regarding depth is that the proportion of numerical semigroups of genus g which have depth $q \leq 3$ tends to 1 as g tends to infinity. This was conjectured by Zhao in [14] and proved by Zhai in [13]. In both cases depth was not mentioned as such but in terms of the Frobenius number and the multiplicity.

Next, we will introduce Kunz tuples and show a little bit of the role they play in the study of numerical semigroups.

1.1 Kunz Tuples

Given a numerical semigroup S with multiplicity m, we have that $S + m = \{s + m | s \in S\} \subseteq S$. In fact, m is the smallest natural number with such property. So for every $0 \leq i < m$, there is some $k_i \in \mathbb{N}_0$ such that

$$\{s \in S | s \equiv i \pmod{m}\} = \{k_i m + i, (k_i + 1)m + i, \dots\}$$

Observe that since $0 \in S$ we will always have $k_0 = 0$, but since m is the smallest positive number such that $S + m \subseteq S$ it comes that $k_i > 0$ in all other cases. These k_i allow us to define a feature of a numerical semigroup, its *Kunz tuple*, which will be $(k_1, k_2, \dots, k_{m-1})$. Notice that in the case of $S = \mathbb{N}_0$ the Kunz tuple associated to S will be the empty tuple. We will use $\mathcal{K}$ to denote the map that associates to each numerical semigroup, its Kunz tuple.

For example, going back to Example 1.2, we have that $\mathcal{K}(\langle 4, 5, 6\rangle) = (1, 1, 2)$. To find out why this tuple is as it is, we have to look at the numbers at the top of each column (except the one with the 0) and divide by $m = 4$.

We have $5 = \mathbf{1} \cdot 4 + 1, 6 = \mathbf{1} \cdot 4 + 2, 11 = \mathbf{2} \cdot 4 + 3$, and hence the tuple $(1, 1, 2)$.

The interesting fact about Kunz tuples is that they characterize numerical semigroups entirely. Each numerical semigroup will have its own tuple. Moreover, it is possible to go the other way around. Given a tuple, it is known under which conditions will it be a Kunz tuple. These two facts are reflected in the following theorem.

Theorem 1.12. *[11] The map $\mathcal{K}$ is a bijection between numerical semigroups of multiplicity m and $(m-1)$-tuples of positive integers $k_1, k_2, \dots, k_{m-1}$ such that for each pair of indices $i,)$ for which it makes sense:*

- $k_i + k_j \geq k_{i+j}$,
- $k_i + k_j + 1 \geq k_{i+j-m}$.

Now, it is important to point out that just by looking at a Kunz tuple it is possible to extract some information about the numerical semigroup it belongs to.

Proposition 1.13. *Given a numerical semigroup S and its Kunz tuple $\mathcal{K}(S) = (k_1, \dots, k_n)$ we have that*

- $m(S) = n + 1$;
- $g(S) = \sum_{i=1}^{n} k_i$;

- $q(S) = \max k_i$ *or* 0 *if the tuple is empty;*
- $F(S) = (q-1)m + j$*, where* j *is maximal such that* $k_j = q$ *or* $j = 0$ *if the tuple is empty.*

Proof. Let us begin by the trivial numerical semigroup $\mathbb{N}_0$. Since its Kunz tuple is the empty tuple, we have that its size n and the sum of its elements are both 0. As a consequence, the formulas for m and g are respectively 1 and 0, which is consistent with the fact that $m(\mathbb{N}_0) = 1$ and $g(\mathbb{N}_0) = 0$. The other two cases come directly from the definition and agree with the values $q(\mathbb{N}_0) = 0$ and $F(\mathbb{N}_0) = -1$.
Now, let $S \neq \mathbb{N}_0$ be a numerical semigroup. The case of the multiplicity comes directly from the definition of Kunz tuple, so let us pay attention to the other invariants. By the way we constructed Kunz tuples, we have that $\mathbb{N}_0 \backslash S = \bigcup_{i=1}^{n} \{i, m+i, \dots, (k_i - 1)m + i\}$. Note that for every i the set $\{i, m+i, \dots, (k_i-1)m+i\}$ has k_i elements, so $g(S) = |\mathbb{N}_0 \backslash S| = \sum_{i=1}^{n} k_i$. Now, recall that the Frobenius number is the greatest integer that does not belong to S, and since $S \neq \mathbb{N}_0$, this translates to $F(S) = \max(\mathbb{N}_0 \backslash S) = \max_{1 \leq i \leq n}((k_i - 1)m + i)$. Now, with what we have let us look at depth.
By definition $q(S) = \left\lceil \frac{c(S)}{m(S)} \right\rceil = \left\lceil \frac{F(S)+1}{m(S)} \right\rceil$, changing $F(S)$ by the previous expression using the Kunz tuple we get to $q(S) = \left\lceil \frac{\max_{1 \leq i \leq n}((k_i-1)m+1)}{m(S)} \right\rceil = \max_i(k_i - 1) + \left\lceil \frac{i+1}{m} \right\rceil = \max_i k_i$. The expression $\left\lceil \frac{i+1}{m} \right\rceil$ cancels to 1 because $0 < i + 1 \leq m$. With this new expression for q we can now rewrite F, and we get the desired $F(S) = (q-1)m + j$, where j is maximal such that $k_j = q$. □

Given a Kunz tuple, if we look at its elements as letters and the whole tuple as a word we will get a Kunz word. Then, Kunz languages, which are the object of study in this thesis, are nothing but collections of such words. Before getting to them, in the next chapter we will talk about the basics of formal languages that we need and then we will use them to introduce properly Kunz languages in Chapter 3.

Chapter 2

Formal Languages and Automata

This chapter is devoted to include all the basics of formal languages and automata that are needed to properly understand the results about Kunz languages. In the first section, we will present some notions of formal languages that will be applied in the study of numerical semigroups. Then, in the second section, we will start by discussing how formal languages are studied using grammars and automata. Firstly we will talk about grammars and how they define a hierarchy in languages and then we will talk about how automata can also be used for this purpose and show their equivalences. The proofs of some results regarding which languages are recognized by each machine have not been included and specific references where they can be consulted have been given instead.
In general, everything in this chapter follows [7], but [12], [5] and [8] have also been consulted in order to explore other points of view and expand ideas.

2.1 Formal languages

As one intuitively may think, a formal language is just a set of words formed with letters from an alphabet.

Given a nonempty finite set of symbols Σ, we denote by Σ^* the free monoid of Σ, i.e. the set containing all possible results concatenating a finite number of symbols of Σ along with the empty word, which will be denoted by λ.
Then, a language L over an alphabet Σ is a subset of Σ^*, that is, a language is a set of words formed with letters in Σ. Sometimes, we will refer to all possible nonempty words over an alphabet for which the notation Σ^+ will be used, in other words, $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$.

Let us illustrate the concept of language with some simple examples. Let $\Sigma = \{a\}$ be the alphabet that just contains the letter a. In this case, we will have that $\Sigma^* = \{\lambda, a, aa, aaa, \dots\}$. That is, all the strings formed by any number of letters a and the empty string λ. Over this alphabet we can define languages such as $\{\lambda\}$, the language formed by the empty string, $\{a\}$ the language formed by the string a or $\{a, aa, aaaa\}$ among infinitely many others. Moreover, we can think of infinite languages, for example $\{a^i | i \in \mathbb{N}_0\}$ or $\{a^{i+j} | i, j \text{ primes}\}$. Although all these examples are over the same alphabet, the simplest one since it only contains one symbol, it seems natural to think that finite languages are somehow less complex than infinite ones and that, even in that case, the last example is more complex than the previous one. Through the rest of the section, we will define, formalize and study this complexity which will be used later to classify families of numerical semigroups in the same complexity scale.

2.1.1 Grammars and Automata

There are essentially two ways to describe languages. The first one is giving a set of rules that explain how to build the strings in that language, what is known as grammars. The other way to describe a language is to recognize which words belong to the language and which do not. This is made by defining a machine able to do this job. These machines are known as automata and we will see that there are some equivalences between languages defined by grammars and languages defined by automata.

2.1.1.1 Grammars

Definition 2.1. A grammar G is defined as a quadruple $G = (V, T, S, P)$ where:

- V is a finite set of objects called *variables*,
- T is a finite set of objects called *terminal symbols*,
- $S \in V$ is a special symbol called the *start variable*,
- P is a finite set of *productions* of the form $x \to y$ (read x produces y), where $x \in (V \cup T)^+$ and $y \in (V \cup T)^*$.

such that V and T are nonempty and disjoint.

The most important part of a grammar is its set of productions, since the production rules are the way in which strings are produced from the start symbol S. Productions are

what makes a difference in complexity between two grammars, depending on the type of rules we will classify grammars into different classes of complexity which will result in a complexity classification in languages.

Production rules are applied in the following way. If we have a production $x \to y$ and a word $w = uxv$, our production is applicable to w and lets us replace x by y in w obtaining the new word $z = uyv$. This is written as $w \Rightarrow z$ and we say that w *derives* z or that z is derived from w.

A production can be applied as many times as desired as long as it is applicable, and can be combined with other rules as well. So if there is a production or set of productions such that $w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$ we say as well that w_1 derives w_n and in this case we write $w_1 \Rightarrow^* w_n$. That is $\Rightarrow^*$ denotes an unspecified non-negative number of productions applied on w_1 to obtain w_n.

Grammars are a tool to define languages. Given a grammar, we can consider all the possible words derived from the start symbol S.

Definition 2.2. Let $G = (V, T, S, P)$ a grammar. The set

$$L(G) = \{w \in T^* | S \Rightarrow^* w\}$$

is the *language generated by* G.

Example 2.1. Let $G_1 = (V, T, S, P)$ with $V = \{S\}$, $T = \{a\}$ and the set P is formed by the production $S \to a$. In this case, the language generated by G is $L(G) = \{a\}$ since there is only one production rule that can be applied exactly once. We start with S, apply the rule $S \Rightarrow a$, and from a we cannot derive anything else.

Example 2.2. Let $G_2 = \{\{S\}, \{a\}, S, \{S \to aS, S \to \lambda\}\}$. Starting with S, we can either apply $S \to \lambda$ and derive λ or apply $S \to aS$ and derive aS. If we repeat the later n times we will end up with $a^n S$ and at that point by using the rule $S \to \lambda$ we get a^n. So we have that $L(G_2) = \{a^i \mid i \in \mathbb{N}_0\}$.

During the 1950s Noam Chomsky introduced a hierarchy of four grammars each of which had more restricted production rules than the previous one. This hierarchy is known as the *Chomsky* or *Chomsky-Schützenberger* hierarchy due to the important role that Marcel-Paul Schützenberger also played in the development of formal languages. As each grammar produces a language, this hierarchy of grammars translates to a hierarchy in languages. Through the section, we will study the different grammars and the languages

they produce as well as its equivalence in automata. Before getting to the definition and results concerning this hierarchy one can look at the following table which summarizes almost everything in the section.

Grammar	Language	Automaton
Type-3	Regular	Deterministic finite accepter
Type-2	Context-free	Nondeterministic pushdown automaton
Type-1	Context-sensitive	Linear bounded nondeterministic Turing machine
Type-0	Recursively enumerable	Turing machine

Table 2.1: Chomsky hierarchy and the relation between languages, automata and grammars.

Remark 2.3. Although in the table grammars are named as *Type-0* to *Type-3* which are the names given by Chomsky at the time, we will usually refer to them by the same name as the language they generate since it is the usual notation in most of the computer science books. There is though the exception of Type-0 grammars which are usually referred to as unrestricted grammars.

Definition 2.4. A grammar $G = (V, T, S, P)$ is said to be *right-linear* if all the productions are of the form $A \to xB$ or $A \to x$ where $A, B \in V$ and $x \in T^*$.
Analogously, G is called to be *left-linear* if all productions are of the form $A \to Bx$ or $A \to x$.
A *regular grammar* is a grammar that is either right-linear or left-linear.
A language L is *regular* if there exists a regular grammar G such that $L = L(G)$.

If we look again at Examples 2.1 and 2.2, we can see that both of them are right-linear grammars and as a consequence the languages they generate are regular.
Regular languages are the simplest ones in the Chomsky Hierarchy, and as we said at the beginning of the section, it seems reasonable to think that finite languages should be considered simpler than infinite ones. In Example 2.2 we have already seen a regular language that is infinite, so it would make sense that all finite languages were regular. This is true and it is not hard to prove as can be seen in the following proposition.

Proposition 2.5. *All finite languages are regular.*

Proof. Let $L = \{w_1, w_2, \dots, w_n\}$ be a language over an alphabet Σ. We need to find a regular grammar G such that $L(G) = L$. This can easily be done by setting the grammar

as $G = (V, T, S, P)$, where $V = \{S\}, T = \Sigma$, and the set P consists of the productions $S \to w_1, S \to w_2, \ldots, S \to w_n$. □

Now, by relaxing the conditions the production rules must fulfil, we can get stronger grammars that allow us to generate more languages. In the case of regular grammars, productions have the form $A \to xB$ (or Bx) where $A, B \in V$ and $x \in T^*$. Now, if we let the right hand of the production be any element of $(V \cup T)^*$ we will get what is known as context-free grammars.

Definition 2.6. A grammar $G = (V, T, S, P)$ is said to be *context-free* if all the productions have the form $A \to x$, with $A \in V$ and $x \in (V \cup T)^*$.
A language L is *context-free* if there exists a context-free grammar G such that $L = L(G)$.

Notice that any element of the form xA or Ax with $A \in V$ and $x \in T^*$ is contained in $(V \cup T)^*$, so any regular grammar is a context-free grammar as well but not the other way around. For example, a production such as $S \to aSa$ is allowed in context-free grammars but not in regular ones. So we have that regular languages are contained in the set of context-free languages. In fact, the inclusion is strict, and the following example of language is context-free but not regular. We will not prove this fact, but it is easy to do so using the *Pumping Lemma* from an automata approach.

Example 2.3. Let G be the grammar $G = (\{S\}, \{a, b\}, S, P)$, with P containing the productions $S \to aSa, S \to bSb$ and $S \to \lambda$. Then G is a context-free grammar.
We can do derivations such as $S \Rightarrow^* aSa \Rightarrow^* abSba \Rightarrow^* abbSbba \Rightarrow^* abbbba$. It is not hard to see that we can produce the empty word, and any word formed with as and bs as long as it is a palindrome. So $L(G) = \{ww^R | w \in \{a, b\}^*\}$

Now, as we have done previously, we can loosen up once more the conditions a grammar has to meet and obtain context-free languages.

Definition 2.7. A grammar $G = (V, T, S, P)$ is called *context-sensitive* if all the productions are of the form $x \to y$, where $x, y \in (V \cup T)^+$ and $|x| \leq |y|$.
A language L is *context-sensitive* if there exists a context-sensitive grammar G such that $L = L(G)$.

Remark 2.8. Equivalently, a context-sensitive grammar can be defined as one whose production rules have the form $uAv \to uwv$ where $u, v \in ((V \cup T) \backslash \{S\})^*$ and $w \in ((V \cup T) \backslash \{S\})^+$. With this alternative definition, it is makes more sense why they are called context-sensitive.

We have that a production looks like $uAv \to uwv$, here u and v constitute the context and what we can or cannot do with A depends on it. The equivalence between both definitions can be consulted in [8, p.6].

Finally, to get to the top of the Chomsky hierarchy we just have to relax completely the conditions that a grammar must fulfil. That is, we do not put restrictions at all, and hence the name.

Definition 2.9. A grammar $G = (V, T, S, P)$ is said to be *unrestricted* if all the productions are of the form $u \to v$, where $u \in (V \cup T)^+$ and $v \in (V \cup T)^*$.
A language L is *recursively enumerable* if there exists an unrestricted grammar G such that $L = L(G)$.

2.1.1.2 Automata

Automata are essentially abstract objects representing the idea of digital computers. With them, we formalize the concept of machine and computation which enables us to study the capabilities of computers.
To begin with, let us study the simplest of automata, deterministic finite accepters or deterministic finite automata. These are models useful to describe computers with a very limited amount of memory.

Definition 2.10. A deterministic finite accepter or dfa is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of *states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

This definition synthesizes the following: the machine consists of a finite number of states (Q), and it is initialized at q_0. Then, at each state, when given an input from the alphabet Σ, it moves to another state according to the transition function δ and when no more inputs are received, it can either land in a final or a non final state.
It is useful to view a dfa as a graph where each vertex is a state and oriented labeled

edges represent the transition function. These graphs are known as *transition graphs*, in them start symbols tend to be marked with an arrow pointing to them, non final states are drawn as circles and final states with double circles. Next, we can see an example of dfa with its transition graph.

Example 2.4. Automaton $M = (Q, \Sigma, \delta, q_0, F)$ and its transition graph, where

- $Q = \{q_0, q_1\}$
- q_0 initial state
- $\Sigma = \{1\}$
- $\delta(q_0, 1) = q_1, \delta(q_1, 1) = q_0$
- $F = \{q_1\}$

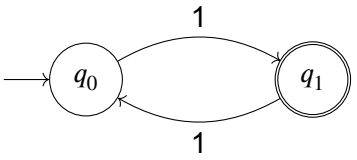


Figure 2.1: Transition graph for M

As it has been said previously, automata can be used to describe languages by recognizing words, and our interest in automata in this thesis relies exactly on this fact.
Given an alphabet Σ and a dfa M whose input alphabet is Σ, we can think of M as reading a word $w \in \Sigma^*$ by initializing it in the start state q_0 and feeding one letter at a time as input. After reading w if M ends up in a final state we say that M accepts w.
More formally this can be defined by extending the transition function δ to $\delta^* : Q \times \Sigma^* \to Q$ recursively:

$\delta^*(q, \lambda) = q$ and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for all $q \in Q, w \in \Sigma^*, a \in \Sigma$.

Then we say that the dfa M with input alphabet Σ and set of final states F accepts $w \in \Sigma^*$ if $\delta^*(q_0, w) \in F$. This allows to define a language as those words accepted by a dfa. That is, given M a dfa, the language defined by M is $L(M) := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$. For example, in the case of Example 2.4, we have that $L(M) = \{1^{2i-1} | i \in \mathbb{N}\}$ the language of non empty strings formed by an odd number of 1s.

Notice that dfa are called *deterministic*. This is to distinguish them from *nondeterministic* finite accepters, or nfa. At this point it is worth talking about nondeterminism and why does it play a role in computation theory. As we have said at the beginning of the chapter, automata are a theoretical model for digital computers, and we know for a fact that computers are deterministic, at each state, given a particular input, its next state is completely determined. So at first, it does not seem very natural to think about nondeterminism, but actually, there are some moments where it does make a lot of sense. We need to look at nondeterminism as a machine parallelizing its calculations. When executing some kind of

algorithm or when modelling some problems it is common to reach some point where a decision has to be made, and it might not be possible to know which is the best one. At this moment, if we add nondeterminism, we allow the machine to explore several options at the same time, that is, we allow it to explore each decision and its path. An example of this practice is what is known as search-and-backtrack algorithms. In the context of language theory, nondeterminism is useful because it provides us more expressiveness, allowing us to work with some trains of thought that determinism does not permit. Sometimes, as in the case of dfa and nfa, which we will see next, it happens that even though the possibilities of each kind of machine differ, the kind of languages that they define is the same, regular languages. So in particular, nondeterminism helps us to reason about regular languages with more powerful tools.
Let us now define nfa, and see how they are indeed equivalent to dfa.

Definition 2.11. A nondeterministic finite accepter, or nfa, is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0 and F are exactly as in dfa, and δ is a function with states and letters as inputs and sets of states as outputs, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ denotes the power set of Q.

So, in summary, an nfa is like a dfa but it is allowed to compute several options at the same time. This is why the outputs of δ are sets, at each of the states of the output it is as if we had a dfa executing that path.
Sometimes, in the definition of an nfa, the domain of the transition function δ is set as $Q \times (\Sigma \cup \{\lambda\})$ permitting λ-transitions. These transitions mean that we let the dfa change state without consuming any symbol. We can use λ-transitions for greater levels of expressiveness and simplicity. Imagine we have an nfa and it is not clear whether at some point the machine should be in q_n or q_m, by setting a λ-transition between those states we allow the nfa to be at both at the same time. Even though, it is a powerful tool, the family of languages defined by both kinds of nfa are the same, we will mention this fact later in detail.
Now we need to look at how a language is defined by a nfa. In this case, in order for a dfa to accept a word it is required that the set of states at which the machine would end when given an input contains at least one final state from F, i.e., we require that at least one of the paths the machine explores leads to a final state.
As we did in the case of dfa, we will extend δ to δ^* and define a language recognized by an nfa from there. Given a dfa $M = (Q, \Sigma, \delta, q_0, F)$ we define $\delta^* : Q \times \Sigma^*$ recursively as

$\delta^*(q,\lambda) = \{q\}$ for every $q \in Q$ and $\delta^*(q,wa) = \bigcup_{q' \in \delta^*(q,w)} \delta(q',a)$ for all $w \in \Sigma^*$, $a \in \Sigma$. Then the language generated by M is $L(M) = \{w \in \Sigma^* | \delta^*(q_0,w) \cap F \neq \emptyset\}$.

Example 2.5. Let $M = (\{q_0,q_1,q_2,q_3,q_4,q_5\},\{a\},\delta,q_0,\{q_3,q_5\})$ with δ defined as

$\delta(q_0,a) = \{q_1,q_4\}, \delta(q_1,a) = \{q_2\}, \delta(q_2,a) = \{q_3\}, \delta(q_3,a) = \emptyset, \delta(q_4,a) = \{q_5\}, \delta(q_5,a) = \{q_4\}$

Notice that for the state q_3 we have that $\delta(q_3,a) = \emptyset$ something that was not possible in deterministic finite accepters. As in the case of dfa, we can draw a transition graph representing the machine:

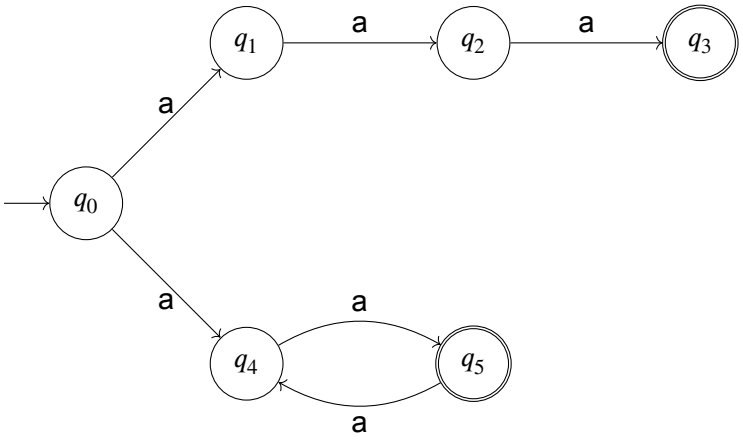


Figure 2.2: Transition graph for M

Looking at the graph may help to find out which language describes $L = L(M)$ accepts the nfa. It is not hard to see that $L(M) = \{aaa, (aa)^i | i \in \mathbb{N}\}$.

If we have a dfa $M_D = (Q,\Sigma,\delta,q_0,F)$, we can build an nfa $M_N = (Q,\Sigma,\delta',q_0,F)$ where $\delta'(q,a) = \{\delta(q,a)\}$ for every $q \in Q$ and every $a \in \Sigma^+$, and then let all λ-transitions go to the empty set $\delta'(q,\lambda) = \emptyset$ for every $q \in Q$. It is clear that $L(M_D) = L(M_N)$. That is, for every dfa there exists an nfa accepting the same language.

Next we see the converse, how for any language defined by an nfa $M_N = (Q_N,\Sigma,\delta_N,q_0,F_N)$, there is a dfa $M_D = (Q_D,\Sigma,\delta_D,\{q_0\},F_D)$ that defines the exact same language. The idea is that the states of M_D are sets of states from M_N, and its final states are those sets that contain at least one element from F_N. Through the proof of the following theorem, it is important to keep in mind that a state in M_D is an element of Q_D but it is a subset of Q_N. Concerning the transition function, we have that δ_N brings elements from Q_N to subsets of Q_N, but with δ_D we need sets in both the domain and the codomain, so we send a set $q \subseteq Q_N$ to the union of all images by δ_N of every single element of q.

Theorem 2.12. *Let $M_N = (Q_N,\Sigma,\delta_N,q_0,F_N)$ be an nfa. Then there exists a dfa $M_D = (Q_D,\Sigma,\delta_D,\{q_0\},F_D)$ such that $L(M_N) = L(M_D)$.*

Proof. In this proof, we will essentially define explicitly a dfa M_D and then prove that it does the same as the nfa M_N.

Let $Q_D = \mathcal{P}(Q_N)$, $F_D = \{q \in Q_D | q \cap F_N \neq \emptyset\}$. Then we define the transition function as $\delta_D(q,a) = \bigcup_{p \in q} \delta_N(p,a)$ for $q \in Q_D$ and $a \in \Sigma$.

We will see now, that everything works as it should. We want to prove that $L(M_D) = L(M_N)$ what by definition is equivalent to prove that $\delta_D^*(\{q_0\}, w) = \delta_N^*(q_0, w)$ for every $w \in \Sigma^*$. We will proceed by induction over the length of words $|w|$.

The initial case $|w| = 0$ is that of $w = \lambda$. In this case $\delta_D^*(q_0, \lambda) = \delta_D(q_0, \lambda) = \cup_{p \in \{q_0\}} \delta_N(p, \lambda) = \delta_N(q_0, \lambda) = \delta_N^*(q_0, \lambda)$.

At his point, by induction, we assume that for any $w \in \Sigma^*$ with $|w| = n$ the statement is true.

Now, let $w \in \Sigma^*$ with $|w| = n+1$. There have to exist $a \in \Sigma$ and $u \in \Sigma^*$ such that $|u| = n$ and $w = ua$. Then $\delta_D^*(\{q_0\}, w) = \delta_D^*(\{q_0\}, ua) = \delta_D(\delta_D^*(\{q_0\}, u), a) \underset{ind.}{=} \delta_D(\delta_N^*(q_0, u), a) \underset{(2)}{=} \bigcup_{p \in \delta_N^*(q_0,u)} \delta_N(p,a) \underset{(2)}{=} \delta_N^*(q_0, ua) = \delta_N^*(q_0, w)$

In (1) we used the definition of δ_D and in (2) the definition of δ_N^*. □

Remark 2.13. The proof of Theorem 2.12 can be extended to take into account lambda transitions in the case nfas had been defined with them. This can be seen in [12, p. 55]. Notice that in the proof there might be many unnecessary states of the dfa defined whose image in the transition function is the empty set. This proof gives just certainty of the equivalence by describing a dfa capable of recognizing the same language but does not provide the best one computationally speaking. It is interesting to point out that converting an nfa to an equivalent dfa is a task that plays an important role in computer science and for which several algorithms have been proposed.

Next, we will see that the family regular languages is the family of languages recognized by dfas. To do so, we will use nondeterminism, which makes the proof easier. This an example of what has been said previously concerning the advantages of using nondeterminism.

Theorem 2.14. *A language L is regular if and only if there exists a dfa M such that $L = L(M)$.*

Proof. By definition a language L is regular if there is a regular grammar G such that $L = L(G)$. This grammar can be either right linear or left linear but in the proof we will only take into account the right linear case since the other one is completely analogous.

Then, as we have seen in Theorem 2.12 it is equivalent to prove the theorem for an nfa that is what we will do since it makes things much easier. So essentially we will prove that given a right linear grammar G there is an nfa M that recognizes the same language and vice-versa.

($\leftarrow$) Let $M = (Q, \Sigma, \delta, q_0, F)$ be any nfa. We can define a grammar $G = (V, T, S, P)$ so that its variables are states of M and its terminal symbols are the alphabet symbol of M. Explicitly, the grammar G is set as $V = Q, T = \Sigma, S = q_0$ and its productions are $P = \{A \to xB | B \in \delta(A, x)\} \cup \{A \to \lambda | A \in F\}$.

By definition we have that $S \Rightarrow xB$ if and only if $B \in \delta(q_0, x)$. Now, if $w = w_1 w_2 \cdots w_n \in \Sigma^*$ with $|w| = n$, we have that $B \in \delta^*(q_0, w)$ if and only if $B \in \delta(\delta(\dots \delta(q_0, w_1), \dots), w_{n-1}), w_n)$. In other words, $S \Rightarrow^* wB$ if and only if $B \in \delta^*(q_0, w)$. We want to see that for every $w \in \Sigma^*$ we have that $S \Rightarrow^* w$ if and only if $\delta^*(q_0, w) \cap F \neq \emptyset$. Notice that $S \Rightarrow^* w \iff S \Rightarrow^* wB \wedge (B \to \lambda) \in P \iff B \in \delta^*(q_0, w) \wedge B \in F \iff \delta^*(q_0, w) \cap F \neq \emptyset$ as we wanted to prove.

($\rightarrow$) This other implication is very similar in nature. Given $G = (V, T, S, P)$ we define $M = (Q, \Sigma, q_0, F)$ with $Q = V, \Sigma = T, q_0 = S, F = \{A \in V | (V \to \lambda) \in P\}$ and for every $x \in \Sigma$ we define $\delta(A, x) = B$ for every B such that $(A \to xB) \in P$. Now following the same steps as before but in the other direction we can get to the result. □

Now, we can improve a dfa, by adding some memory to it, having then a model of a more complex automaton.

Remark 2.15. By $\mathcal{P}_f(A)$ we will denote the finite power set of a set A. In other words, $\mathcal{P}_f(A)$ is the set of all finite subsets of A.

Definition 2.16. A *nondeterministic pushdown accepter* or *ndpa* for short, is a septuple $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ where

- Q is i a finite set of internal states,
- Σ is the input alphabet,
- Γ is a finite set of symbols called the *stack alphabet*,
- $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to \mathcal{P}_f(Q \times \Gamma^*)$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $z \in \Gamma$ is the *stack start symbol*,

- $F \subseteq Q$ is the set of final states.

There are two aspects that make a difference between nondeterministic pushdown automata and deterministic finite accepters. The first one is the memory and how it is used, and the second is nondeterminism.
In a ndpa the memory model is a stack that can be filled with symbols from the stack alphabet Γ and that is initialized by a start symbol $z \in \Gamma$. Then, at each step, the automaton can read and remove the top symbol of the stack and then write a string of symbols one by one on top of it pushing all the previous symbols of the stack down. We can imagine the memory as a stack of plates where each plate has a symbol written, we are only allowed to remove a plate from the top and then add some other plates on top of the stack. This type of memory access is usually referred to as *last in first out*. It is common as well to use *pushing* and *popping* as synonyms of writing and reading symbols respectively.
The other aspect is nondeterminism. One might think that as in the case of dfas and nfas, nondeterminism will not change the kind of language that the machine is able to recognize, but this is not true in the case of pushdown automata. There is, in fact, a deterministic version in which the transition function is similar to that in dfa, but it can be seen that this type of pushdown automaton is not able to recognize the same languages as the nondeterministic one.

Now we need to look at how a language is defined by an ndpa. In order for an ndpa to accept a word it is required that the set of states at which the machine would end when given an input contains at least one final state from F, i.e., we require that at least one of the paths the machine explores leads to a final state as in the case of dfas. The thing though, is that since with npdas we have a memory stack that conditions the movements through states, extending δ to δ^* is not that easy, and its notation might be confusing and not much useful to do mathematical arguments. For this reason, we introduce the notion of instantaneous description which is very handy to keep track of what an ndpa is doing at a precise moment.
Given an npda $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, an *instantaneous description* of M is a triplet (q, w, u) where q is the current state of M, w is the unread part of the input string, and u is the content of the stack (with left to right in u indicates top to bottom in the stack). So with an instantaneous description, we can express the position of an ndpa. Now we need to describe how does M move from an instantaneous description to another. For

$a \in \Sigma, b, y \in \Gamma, x \in \Gamma^*$ and $w \in \Sigma^*$ a move from (q_1, aw, bx) to (q_2, w, yx) is possible if and only if $(q_2, y) \in \delta(q_1, a, b)$. In this case, we denote $(q_1, aw, bx) \vdash (q_2, w, yx)$. So we have that $(q_1, aw, bx) \vdash (q_2, w, yx)$ indicates a single step in M; moves with an arbitrary number of steps are denoted by $(q_1, aw, bx) \vdash^* (q_2, w, yx)$ similarly to how derivations in grammars are denoted. With this simple notion, we can already define how is the language accepted by M. It is the set $L(M) = \{w \in \Sigma^* : (q_0, w, z) \vdash^* (p, \lambda, u), p \in F, u \in \Gamma^*\}$. In plain English, it is the set of words that when fed to M as input make the machine stop in a final state. Note that we need to see this in the nondeterministic way in which several ways are explored in an identical way as in nfa. It is important as well to point out that in order for a word to be accepted it just needs to end in a final state. Some authors include the additional requirement that the stack has to be empty. Both definitions are equivalent in the sense that they accept the same family of languages. Each one has its own benefits, for example, the one we gave allows a more general and flexible model of computation and is better to deal with proofs while the other one simplifies the construction of the deterministic version of the machine: the deterministic pushdown automaton.

The following theorem states which is the family of languages recognized by ndpas. The proof is not included but each of the implications of the theorem correspond to [7, Theorem 7.1 p.188] and [7, Theorem 7.2 p.195]

Theorem 2.17. *A language L is context-free if and only if there exists an ndpa M such that $L = L(M)$.*

As in any kind of automata, ndpas have a way to be represented by a transition graph. In the very same way as we did with dfas, we draw circles to represent states, final states are marked by a second circle, and the initial state has an arrow pointing at it. Finally, moves are represented by arrows labelled with three symbols, the first one indicates what is being read from the input, the second what is being read from the stack and the third represents the string to be inserted in the stack. Let us see an example to clarify all the contents about nondeterministic pushdown automata.

Example 2.6. Consider an ndpa $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ where $Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}$, $\Gamma = \{0, 1\}$, $z = 0$, $F = \{q_3\}$ and δ defined as

$$\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}, \qquad \delta(q_1, b, 1) = \{(q_2, \lambda)\},$$
$$\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}, \qquad \delta(q_2, b, 1) = \{(q_2, \lambda)\},$$
$$\delta(q_1, a, 1) = \{(q_1, 11)\}, \qquad \delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}.$$

Notice that it is indeed nondeterministic since $\delta(q_0, a, 0)$ is a set containing two elements, so the automaton has to explore two different ways at that point. Pay attention as well to the fact that some transitions are not specified, meaning that would go to the null set representing a dead configuration. Using the kind of transition graphs we described, we can represent M as follows:

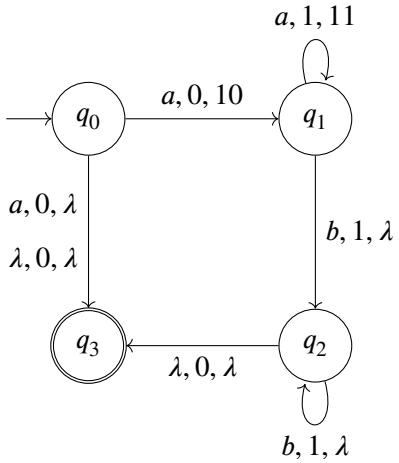


Figure 2.3: Transition graph of an ndfa that recognizes $L = \{a^n b^n : n \in \mathbb{N}_0\} \cup \{a\}$.

This npda essentially does the following, when it finds an a it adds a 1 to the stack and repeats this process until a b is found. Then, when the b is found, the machine removes a 1 from the stack and keeps doing it. So if the input is finished and the stack is empty it ends in the final state q_3. This can happen as well if the input consists of $w = a$. So the language recognized by this ndpa is $L(M) = \{a^n b^n : n \in \mathbb{N}\} \cup \{a\}$. Notice that if at the beginning, in state q_0, the input consists of a b the move is not defined, meaning it falls into a dead configuration thus rejecting the word. The same happens if after the first b in state q_1 an a is found or if at any time the input has ended and the stack is not empty or the stack is empty but the input has not completely been read. This example can be found in [7, p.181], as well as some other examples.

Finally, it is time to get to the most complex kind of automaton, the *Turing machine*. This increase in complexity, or power of computation, is achieved by improving the kind of memory and how it is accessed. In a Turing machine, instead of a stack, we have a tape divided into cells in each of which a symbol can be written. The machine has a head that moves through the tape and at each time it can read a symbol, write another one and move one cell left or right.

Definition 2.18. A Turing machine M is defined as a tuple $M = (Q, \Sigma, \Gamma, q_0, \square, F)$ where

- Q is the set of internal states,
- Σ is the input alphabet,
- Γ is a set of symbols called the *tape alphabet*,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $\square \in \Gamma$ is a special symbol called *blank*,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,

In addition, we require $\Sigma \subseteq \Gamma \backslash \{\square\}$. The elements of the set $\{L, R\}$ represent the head of the machine moving left (L) or right (R) along the tape after having written a symbol on it.

Remark 2.19. The transition function δ is actually a partial function, that is, its domain is $A \subseteq Q \times \Gamma$. However we can always turn δ into a total function by adding a non final sink state so that for every originally undefined move, when reading the symbol on the tape the machine moves to the sink state and then every symbol read makes the machine stay in that sink state.

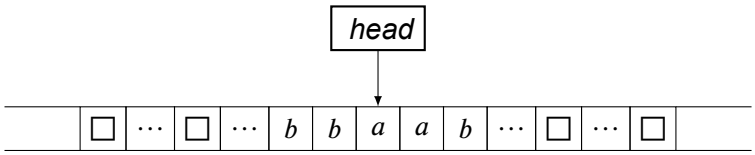


Figure 2.4: Model of a Turing Machine.

Now, as in the previous models of automata, we need to define how a Turing machine accepts a language. In this case, just like with ndpas, the most convenient way to do so is through instantaneous descriptions. Notice that the configuration of a Turing machine at any time is completely determined by the position of the head at that time, the current state, and the contents of the tape. To represent this we will use the notation $x_1 q x_2$ with $x_1, x_2 \in \Gamma^+$ and $q \in Q$ to represent that the current state is q, that the content of the tape is $x_1 x_2$, with anything left or right of that being blank symbols ($\square$), and the head is positioned over the cell containing the leftmost symbol of x_2.

Once we have defined positions, it is time to talk about moves. A move from a configuration to the next one, similarly to how it is done in ndpas in Page 24 will be denoted by $\vdash$,

that is, if we have $\delta(q_1, a) = (q_2, b, R)$, then the move $xq_1ay \vdash xbq_2y$ is made whenever the current state is q_1 and the content of the tape is xay with the head being positioned over the cell containing the symbol a. In order to denote an arbitrary number of moves we will use $\vdash^*$ as previously.

Now we can easily define how a language is defined by a Turing machine. The idea is that if we start the machine in its initial state with a string w in its tape and ends up halting in a final state then w is accepted, otherwise it is rejected. Let us define this formally:

Definition 2.20. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine. The language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \vdash^* x_1 q_f x_2 \text{ for some } q_f \in F \text{ and } x_1, x_2 \in \Gamma^* \right\}$$

The following theorem states which is the family of languages recognized by Turing machines. The proof is not included but each of the implications of the theorem correspond to [7, Theorem 11.6 p.283] and [7, Theorem 11.7 p.284]

Theorem 2.21. *A language L is recursively enumerable if and only if there exists a Turing machine M such that $L = L(M)$.*

Remark 2.22. When giving a string as input to a Turing machine, there are three possible outcomes: it halts in a final state, it halts in a non final state or it does not halt at all, that is the machine enters an infinite loop. We have defined recursively enumerable languages to be those in which every word that does not halt in a final state is rejected. There exists as well the family of *recursive languages*, that is defined by just rejecting words that halt in a non final state, in other words, strings that make the machine enter an infinite loop are accepted as well. Recursive languages are also known as *decidable languages* or *Turing-decidable*. At the same time, recursively enumerable languages are called *Turing-recognizable* or simply *recognizable*.
The family of decidable languages strictly contains the family of recognizable languages but it is not included in the Chomsky hierarchy.

In order to visually represent a Turing machine in a similar way to what we have already done with finite accepters, we can as well draw transition graphs. The design is the same for states, each one represented by a circle, the initial state pointed by an arrow and final states marked with a double circle. Now, arrows between states represent the moves and

they are labeled with three symbols, the first one representing the symbol to be read on the tape, the second the symbol to be written and the third representing if the head of the machine has to move left (L) or right (R). Let us now illustrate everything that has been said about Turing machines with an example.

Example 2.7. Consider the Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ defined by $Q = \{q_0, q_1, q_2, q_3, q_4, q_a\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b, c, x, y, z, \square\}$, $F = \{q_a\}$ and the following transition function δ:

$\delta(q_0, a) = (q_1, x, R)$,
$\delta(q_0, y) = (q_4, y, R)$,
$\delta(q_1, y) = (q_1, y, R)$,
$\delta(q_1, a) = (q_1, a, R)$,
$\delta(q_1, b) = (q_2, y, R)$,
$\delta(q_2, z) = (q_2, z, R)$,
$\delta(q_2, b) = (q_2, b, R)$,
$\delta(q_2, c) = (q_3, z, L)$,
$\delta(q_3, a) = (q_3, a, L)$,
$\delta(q_3, b) = (q_3, b, L)$,
$\delta(q_3, y) = (q_3, y, L)$,
$\delta(q_3, z) = (q_3, z, L)$,
$\delta(q_3, x) = (q_0, x, R)$,
$\delta(q_4, y) = (q_4, y, R)$,
$\delta(q_4, z) = (q_4, z, R)$,
$\delta(q_4, \square) = (q_4, \square, L)$.

Just by looking at the definition of the Turing machine, it is not easy to imagine how it is or to get an idea of how it works. In order to fight this, let us look at its transition graph (Figure 2.5).

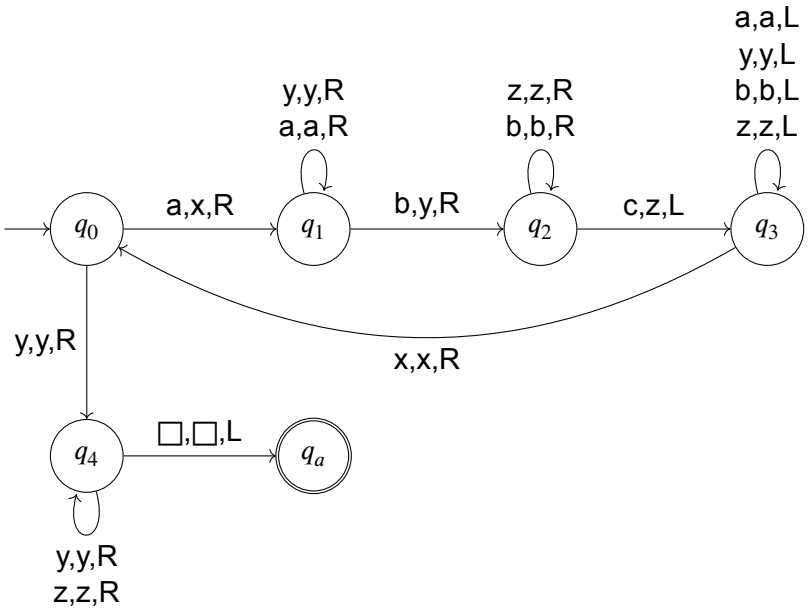


Figure 2.5: Turing Machine that recognizes $L = \{a^n b^n c^n : n \in \mathbb{N}\}$

Instead of trying first to analyze the functioning of the machine and then deduce what language does it recognize, we will do the opposite, since this automaton has been constructed to recognize a specific language, so this order will seem more natural.
The language recognized by this Turing machine M is $L(M) = \{a^i b^i c^i : i \in \mathbb{N}\}$.
The idea behind its functioning is that we start by finding an a and crossing it by changing it into an x then go right until a b is found and change it into a y and next go right until a c is found and marked as crossed by writing a z. Finally, we go all the way back to the beginning of the word and start again doing the same thing. If we repeat this process until no letter is left that means the word has the form $a^i b^i c^i$. On the other hand, if at some point we find a letter that is not the one we are looking for, or we cannot find the letter we are looking for, the word is rejected. In the first case that would mean the word has not the structure $a^i b^j c^k$ and in the second we would have that i, j or k do not represent the same natural number.

The previous example shows the increase in power of Turing machines compared to npda, since the language $L = \{a^i b^i c^i : i \in \mathbb{N}\}$ is not context-free. An explicit prove of this fact can be found in [5, Example 7.19, p.289] or [7, Example 8.1, p.209]. The latter one actually talks about $\{a^i b^i c^i : i \in \mathbb{N}_0\}$ but the same argument is valid. Both of them rely on the use of the Lemma 3.11 about which we will talk later in the next chapter.

To this point, we have seen three different types of machines each of which recognizes a different type of language but there is still a language type missing, context-sensitive languages. The automata that recognize context-sensitive languages are a special kind of Turing machines with a bounded tape and once more with the factor of nondeterminism playing a role. Let us see this step by step.
A nondeterministic Turing machine is an automaton such as the regular Turing machine but where the transition function's set of destination is $\mathcal{P}(Q \times \Gamma \times \{L, R\})$. That is, as in the case of nondeterministic pushdown automata, at each step we allow the machine to explore different paths at the same time. The interesting thing though is that unlike dpa where nondeterminism makes a difference in the kind of language the machine can recognize, there is no such difference when it comes to Turing machines. For any nondeterministic Turing machine M_1, there exists a deterministic one M_2 accepting exactly

the same words, i.e. verifying $L(M_1) = L(M_2)$. In fact, there are many different models of Turing machines where a feature or some features of the regular one are modified and all these are equivalent. Some of these are multi-tape Turing machines where two or more tapes are used simultaneously used or Turing machines with a stay option, where $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$.

Given a nondeterministic Turing machine M the language it recognizes $L(M)$ is defined in the same way as in the other types of nondeterministic automata. That is, a string w is accepted if there is at least a possible configuration that accepts it. We can extend the notation for moves between configurations in order to add nondeterminism just by saying that $xq_1ay \vdash xbq_2y$ if and only if $(q_2, b, R) \in \delta(q_1, a)$. Then the formal definition of $L(M)$ will be the same as for deterministic Turing machines.

Definition 2.23. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a nondeterministic Turing machine. The language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \vdash^* x_1 q_f x_2 \text{ for some } q_f \in F \text{ and } x_1, x_2 \in \Gamma^* \right\}$$

Apart from trying to upgrade Turing machines, we can try to set some limitations. If in a nondeterministic Turing machine instead of having an infinite tape we add a left and right end to it we will get what is known as a *linear bounded automaton*.

Definition 2.24. A linear bounded automaton or lba is a nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ subject to the restriction that Σ must contain two special symbols [and], such that $\delta(q_i, [) \in \mathcal{P}(Q \times \Gamma \times \{R\})$ and $\delta(q_i, [) \in \mathcal{P}(Q \times \Gamma \times \{L\})$. That is, the head of M is not allowed to move left from [or right from] and hence the tape is restricted to the space of the input. Equivalently, we can allow the machine to use a part of the tape that is linearly dependent on the input length, or similarly, allow the tape to have multiple tracks. Note that since the tape is restricted by the input, there might be no need for the $\square$ symbol.

The following theorem states which is the family of languages recognized by lbas. The proof is not included but each of the implications of the theorem correspond to [7, Theorem 11.8 p.288] and [7, Theorem 11.9 p.288]

Theorem 2.25. *A language L is context-sensitive if and only if there exists a linear bounded automaton M such that $L = L(M)$.*

Even though nondeterministic and deterministic Turing machines are equivalent, the same thing cannot be said when the tape is restricted to the input. To date, still remains an open problem if the class of languages accepted by deterministic lbas and nondeterministic lbas is the same. Languages accepted by deterministic lbas are called deterministic context-sensitive and are included in the set of context-sensitive but it is unknown whether the inclusion is strict.

With this, we end the chapter since everything about these subjects has already been presented. In the next chapter, we will apply most of the content of this chapter, paying special attention to linear bounded automata.

Chapter 3

Kunz Languages

The aim of this final chapter is to deal with the main subject of the thesis, Kunz languages. We will start by defining them and showing their connections with semigroups. Right after we will start to classify these languages by applying what has been seen in Chapter 2. When needed, we will introduce lemmas or theorems that will be used right after.

In Section 1.1, it has been shown how given a numerical semigroup S we can associate in a unique way a tuple to it, its Kunz tuple. Now the idea is to see this tuple as a word where letters are the natural numbers that constitute the tuple.

Definition 3.1. A (possibly empty) word $w = w_1 w_2 \cdots w_n$ is a *Kunz word* or simply Kunz, if

- $w_i \in \mathbb{N}$ for all $i \in \{1, 2, \dots, n\}$
- $w_i + w_j \geq w_{i+j}$
- $w_i + w_j + 1 \geq w_{i+j-n-1}$

for all indices i, j for which the inequalities are defined.

So given a semigroup S, and its Kunz tuple $\mathcal{K}(S) = (k_1, k_2, \dots, k_n)$ its associated Kunz word is $w = k_1 k_2 \cdots k_n$ and the same bijection between Kunz words and numerical semigroups seen in Theorem 1.12 holds with the same properties as in tuples. Now, we aim to study the depth of semigroups by looking at words that come from a certain depth.

Definition 3.2. Let $q \in \mathbb{N}_0$ the Kunz language of depth q is

$$K_q = \{w \in \{0, 1, \dots, q\}^* | w_i \text{ is Kunz and } \max(w_i) = q\}.$$

That is, the language formed by all Kunz words of depth q.

Remark 3.3. Although there is not any Kunz word containing zeros, the alphabets over which Kunz languages are defined starting with zero in order to include K_0 in a general definition. This is due to the fact that alphabets need to be nonempty and finite. That is, for K_0 which is the language formed by the empty word, we still need to have some symbol in the alphabet in order for it to be nonempty. And for K_q with $q > 0$ each language needs at least q symbols in the alphabet, but as alphabets have to be finite we cannot use the same alphabet to define all of these languages. So we decided to use the alphabets $\{0, \dots, q\}$ which are generally defined and serve our purposes. Another possible choice would have been defining K_0 over the alphabet $\{0\}$, and K_q over $\{1, \dots, q\}$ for $q > 0$.

From now on, we can already start to work on our goal, classify Kunz languages in the Chomsky hierarchy. Step by step we will determine where does each Kunz language fit in the hierarchy by giving a machine capable to recognize it.
To begin with, we will study K_0, the Kunz language of depth 0. There is only one numerical semigroup with depth 0, the trivial one $\mathbb{N}_0$, and its Kunz word is the empty word λ. So K_0 will be the language consisting of exactly that word. That is, $K_0 = \{\lambda\}$. It is easy to verify that this language is regular, which can be seen presenting a dfa that accepts K_0.

Proposition 3.4. *The Kunz language of depth* 0 *is regular.*

Proof. The following transition graph represents a dfa recognizing K_0.

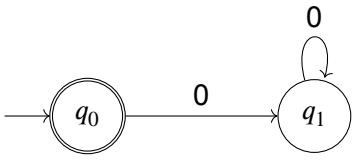


Figure 3.1: dfa that recognizes K_0.

□

Our next step is that of studying K_1, in this case, to think about the words contained in K_1 is easier to rely on the conditions that define a kunz language than to think about semigroups. We are looking at words formed by ones that are not empty. In other words, we have that $K_1 = \{1^i | i \in \mathbb{N}\}$. Once again, with a similar dfa, we can prove that K_1 is regular.

Proposition 3.5. *The Kunz language of depth* 1 *is regular.*

Proof. The following transition graph represents a dfa recognizing K_1.

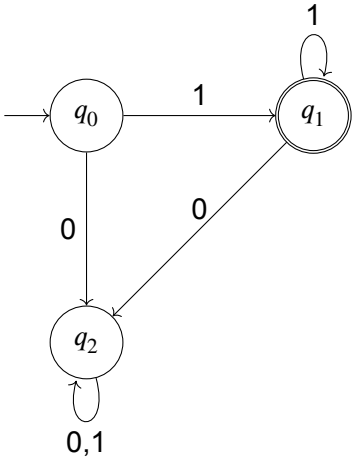


Figure 3.2: dfa that recognizes K_1.

□

The case of K_2 is pretty similar to that of K_1, we have that any nonempty word consisting of ones and twos that contains at least a two will belong to K_2, so $K_2 = \{w \in \{1,2\}^* | 2 \in w\}$, and using a very similar automaton one can prove that this language is also regular.

Proposition 3.6. *The Kunz language of depth* 2 *is regular.*

Proof. The following transition graph represents a dfa recognizing K_2.

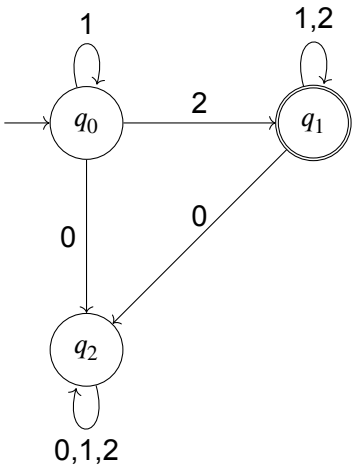


Figure 3.3: dfa that recognizes K_2.

□

The case of K_3 is very different from the ones we have studied until now. Up to this moment, we have proved that languages were regular by giving a dfa that could recognize them. The language K_3 is more complex than the previous ones, and next, we will give

arguments to show that it cannot be of the same type as the previous ones. In fact, at the end of the chapter, we will see that K_3 is actually a context-sensitive language, but first, we will show that it is not any of the previous types in the hierarchy. As we have seen in the previous chapter, every family of languages in the Chomsky hierarchy is included in the next one, so proving that K_3 is not context-free would have been enough. However, every step made during the research has been included in order to show the increase in complexity of the arguments needed in the proofs. Let us start by showing that K_3 is not regular.

Lemma 3.7. *Let n, m and q be positive integers, with $q \geq 3$. Then*

(i) *the word $w = 1^n 2^n 3^n \dots (q-2)^n (q-1)^n q$ is Kunz;*

(ii) *the word $w = 1^{n+m} 2^n 3^n \dots (q-2)^n (q-1)^n q$ is not Kunz.*

Proof. We now prove part (i).
In order for w to be Kunz we have to check that it satisfies the following two conditions.

- $w_i + w_j \geq w_{i+j}$
- $w_i + w_j + 1 \geq w_{i+j-\ell-1}$

for all indices i, j for which the inequalities are defined, being $\ell = |w|$.
For $x \in \mathbb{R}$, let $\lceil x \rceil = \min\{r \in \mathbb{Z} \mid x \leq r\}$. It is clear that $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$.
Note that $\ell = (q-1)n + 1$. Observe also that $w_\kappa = \left\lceil \frac{\kappa}{n} \right\rceil$ for all indices κ. It follows that if i and j are indices such that $1 \leq i + j \leq (q-1)n + 1$, then

$$w_i + w_j = \left\lceil \frac{i}{n} \right\rceil + \left\lceil \frac{j}{n} \right\rceil \geq \left\lceil \frac{i+j}{n} \right\rceil = w_{i+j},$$

thus w satisfies the first Kunz condition.

If i and j are such that $i + j > (q-1)n + 1$, then

$$\left\lceil \frac{i}{n} \right\rceil + \left\lceil \frac{j}{n} \right\rceil \geq \left\lceil \frac{i+j}{n} \right\rceil \geq \left\lceil \frac{(q-1)n+1}{n} \right\rceil = q - 1 + \left\lceil \frac{1}{n} \right\rceil = q.$$

It follows that w also satisfies the second Kunz condition.
Next we prove part (ii). If we take $i = n + 1$ and $j = n + m$, we have that $w_i + w_j = 2$, but $w_{i+j} = w_{2n+m+1} = 3$, so w does not meet the first condition to be a Kunz word. □

Proposition 3.8. *The Kunz language of depth 3 is not regular.*

Proof. Let us suppose, for the sake of contradiction, that K_3 is a regular language. Then, there has to be a dfa M that recognizes it. Let n be the number of states of such dfa.
By Lemma 3.7 the word $w = 1^n2^n3$ is a Kunz word of depth 3, so it has to be accepted by M, but since M has only n states while reading the first n ones of w a M has to come at least twice to the same state q_1 due to the pigeonhole principle. In other words, while reading the ones in w, a cycle $q_1 \to q_2 \to \cdots \to q_i \to q_1$ of length i is walked. As a consequence, any word adding any multiple of i ones will be accepted as well. Then for $i > 0$ the word $1^{n+i}2^n3$ is accepted by M which is a contradiction since $1^{n+i}2^n3$ is not a Kunz word, again by Lemma 3.7. □

This same argument can be adapted to prove that K_3 cannot be recognized by a deterministic pushdown automaton and as a consequence K_3 is not a deterministic context-free language.

Proposition 3.9. *The language K_3 is not a deterministic context-free language.*

Proof. The idea behind this proof is exactly the same as in Proposition 3.8. By contradiction we will suppose that K_3 is indeed recognized by a deterministic pushdown automaton M. Then choosing a word that should be recognized and using the pigeonhole principle we will see that some word that is not in K_3 is recognized as well.
Let M be a deterministic pushdown automata that recognizes K_3, the word $w = 1^s2^s3$ with $s \in \mathbb{N}$ is in K_3. Since the number of states and symbols in the stack is finite, for a sufficiently large s, due to the pigeonhole principle, while reading the ones in w, the automaton M will go twice through the same state q, like this $q \to q_1 \to \cdots \to q_n \to q$.
Moreover, if s is sufficiently large, M will have both times in q the same n symbols on top of the stack hence creating a cycle. So any multiple of n ones can be added to w so that it will be still recognized by M which leads to the contradiction. □

It is worth noting that with the same base as in the previous proofs, one can get to the *Pumping lemma*, a very well-known lemma that is useful to show that a language is not regular and with which Proposition 3.8 can easily be proven. Many examples of the application of this lemma can be found in [7].

Lemma 3.10 (Pumping lemma)**.** *Let L be an infinite regular language. Then there exists $m \in \mathbb{N}$ such that for any $w \in L$ with $|w| \geq m$ can be decomposed as $w = xyz$ with $|xy| \leq m$ and $|y| \geq 1$ such that $xy^iz \in L$ for every $i \in \mathbb{N}_0$.*

The lemma has many versions and generalizations, even one for context-free languages.

Lemma 3.11 (Pumping lemma for context-free languages)**.** *Let L be an infinite context-free language. Then there exists $m \in \mathbb{N}$ such that for any $w \in L$ with $|w| \geq m$ can be decomposed as $w = uvxyz$ with $|vxy| \leq m$ and $|vy| \geq 1$ such that $uv^i xy^i z \in L$ for every $i \in \mathbb{N}_0$.*

With this version of the lemma, though, we were not able to show that K_3 does not belong to the family of context-free languages. There is a generalized version of the pumping lemma for context-free languages known as Ogden's lemma, further generalized by Bader and Moura (see Theorem 3.14).

Lemma 3.12 (Ogden)**.** *Let L be an infinite context-free language. Then there exists $m \in \mathbb{N}$ such that for any $w \in L$ with $|w| \geq m$, we can select m distinguished positions so that w can be decomposed as $w = uvxyz$ with:*

1. *vxy has at most m distinguished positions;*
2. *vy has at least one distinguished position;*
3. *$uv^i xy^i z \in L$ for every $i \in \mathbb{N}_0$.*

Unfortunately, as in the previous case, with Ogden's lemma we were not capable to prove that K_3 is not context-free. For this reason we tried without success using the previously mentioned generalization made by Bader and Moura at the end of the seventies, but we obtained results for $q \geq 5$.

Remark 3.13. Given a language L and a word $w \in L$ we may label some positions (letters) in w as *distinguished* and some others as *excluded*. A particular position can be both distinguished and excluded or maybe neither. When labeling positions of w we will denote $d(u)$ (resp. $e(u)$) as the number of distinguished (resp. $excluded$) positions of a substring u of w.

Theorem 3.14. *[1] For any context-free language L, exists $p \in \mathbb{N}$ such that for every $w \in L$, if $d(w)$ positions are labeled as distinguished and $e(w)$ are labeled as excluded with $d(w) > p^{e(w)+1}$, then there exist u, v, x, y, z such that $w = uvxyz$ and:*

1. *$d(vy) \geq 1$ and $e(vy) = 0$;*

2. $d(vxy) \leq p^{e(vxy)+1}$;

3. $uv^i xy^i z \in L$ *for every* $i \geq 0$.

Using this theorem we can now prove that for $q \geq 5$ the language K_q is not context-free.

Proposition 3.15. *For* $q \geq 5$ *the Kunz language of depth* q *is not context-free.*

Proof. We will argue by contradiction. Suppose that K_q, the Kunz language of depth q, is context-free. Then by Therorem 3.14, there exists a $p \in \mathbb{N}$ satisfying the conditions of the theorem. Let $n = p^q + 1$. By Lemma 3.7, the word $w = 1^n 2^n 3^n \dots (q-2)^n (q-1)^n q$ belongs to K_q.

Now for every $2 \leq m \leq q$ we label the first appearance of m in w as excluded and the positions with the 1s as distinguished. Next we can see w represented with the distinguished positions in green and the excluded ones in red.

$$w = \underbrace{11\cdots1}_{n}\,2\,\underbrace{22\cdots2}_{n-1}\,3\,\underbrace{33\cdots3}_{n-1}\,44\cdots(q-2)(q-1)\,\underbrace{(q-1)(q-1)\cdots(q-1)}_{n-1}\,q$$

We have $d(w) = n = p^q + 1$, and $e(w) = q - 1$, hence $d(w) = p^q + 1 > p^q = p^{e(w)+1}$. As a consequence, the three statements of Theorem 3.14 must hold.

So we will have $w = uvxyz$, where vxy contains at least a 1. Since by the first statement $e(vy) = 0$ we have that neither v nor y contain two different letters, and at least one of them has to be completely made of 1s. So either $vxy = 1^k$ for some $k > 0$ or $v = 1^k$ and $y = m^s$, for some $k > 0$, $2 \leq m < q$ and $s \geq 0$.

In the case $vxy = 1^k$, by the third statement of Theorem 3.14 we could pump v and y and get $1^{n+t} 2^n 3^n \dots (q-2)^n (q-1)^n q$ which is not Kunz, by the proof of Lemma 3.7.

In the other case, $v = 1^k$ and $y = m^s$, we have two options. The first option is that y consists entirely of 2s. If this is so, we could pump y until getting two indices i, j such that $w_i, w_j \leq 2$ but $w_{i+j} \geq 5$.

If y is of the form m^s for some $m \geq 3$ and $s \geq 1$, we could pump v sufficiently until getting two indices i, j such that $w_i = w_j = 1$ but $w_{i+j} \geq 3$. □

Remark 3.16. Note that in Proposition 3.15 in order for the argument to work, the condition $q \geq 5$ is necessary. This is because in the case $v = 1^m$ and $y = 2^m$ the word $uv^\kappa xy^\kappa z$ would still be Kunz. That is, pumping a lot of 2s gets to a non Kunz word in the case of K_5 because $2 + 2 < 5$ but it would not make any difference in the cases of K_3 and K_4 since $2 + 2 \geq 3$, and the condition $w_i + w_j \geq w_{i+j}$ would still hold.

So far, we have seen that K_3 is not regular. In fact, Propositions 3.8 and 3.9 can be adapted to show the same result for depth $q > 3$ by simply changing the word 1^n2^n3 by $1^n2^n3^n \dots (q-2)^n(q-1)^n q$ and using Lemma 3.7. Moreover, we have seen that K_q is not context-free for $q \geq 5$.

Next, we will see how K_3 is context-sensitive by giving an lba capable of accepting it. Since the machine required for this task is pretty complex we will not give a formal definition of it, instead, a high-level description of the automaton will be provided. In this high-level description, we will specify some steps that an lba is capable to do and hence describe its behaviour but not its states and moves. Then, in a similar way, we will describe two kinds of lba capable of accepting any Kunz language K_n for $n \in \mathbb{N}$ and as a consequence prove that all Kunz languages are context-sensitive. In order to make clear that every single step can indeed be done by an lba all the details of every subroutine needed in the other machines have been specified in Appendix A.

Proposition 3.17. *The Kunz language of depth* 3 *is context-sensitive.*

Proof. We give a high-level description of an lba with 5 tracks that accepts K_3. The first track contains the input, and the second and third tracks will be used to store in unary notation indices of 1s in the word w. The fourth track is used to eventually compute and store the result of the sum of the indices of the previous tracks. Finally the last one just stores in unary notation the length of the word w. The amount of tape used will be inferior to $6|w|$ where $|w|$ is the length of the input word. So at the very beginning, the first track will be $\square w_1 w_2 \cdots w_n \square \cdots \square$ and the other tracks completely filled with blanks. The first blank right before w_1 will remain untouched since it marks the left end of the tape. We could substitute it with the symbol $[$ we gave in the definition of lba. And the same happens with the rightmost $\square$, it serves as a $]$ we just used blanks to simplify the notation.

Notice that in order for a word $w = w_1 w_2 \cdots w_n \in \{0,1,2,3\}^*$ to be in Kunz it is necessary in sufficient that $w_i + w_j \geq w_{i+j}$. To contradict this fact, the only possible way is that either w contains a zero, (which is just a technicality), or that $w_i = w_j = 1$ and $w_{i+j} = 3$. Then in order for w to be in K_3 there is the additional condition that $3 \in w$.

The algorithm that the machine will execute is the following: first, check that the word does not contain any zero and that it contains at least a 3. Then from the beginning of the word going left to right, find the first 1, cross it, write its position (i), check if w has a 3 in position ($i+i$), go back to the 1 we just crossed, and check for every other 1 in position j after the x

if the word contains a 3 in position $i + j$. Once all other 1s are verified for this first 1, do the same thing with the others.

The machine is described using steps. These steps are numbered and followed in order unless a step requires to move to another step that is not the following one. We will call for some subroutines such as adding numbers, going a specific number of steps back or forth or comparing natural numbers. After the design of the machine, we will analyze how these functions can be implemented on an lba.

Steps

0. Check if there are any 0s, if so reject, otherwise go back left at the beginning of the tape.

1. Check if there is at least a 3 and go back to the beginning of the tape, otherwise reject. At every step add a 1 on every cell of the fifth track in order to store the length of the word in it.

2. Go right until the first 1 is found and at every step right, write a 1 in the same cell but in the second and third tracks. If there are not any 1s, accept the word. If a 1 is found, change it by an x (cross the 1) and still write 1s in the second and third tracks. At this point, we will have the 1 we are about to check marked with an x and its index i written in the second and third tracks.

3. On the fourth track compute $i + i$ and write it down on the fourth track. Then go to the position with that index ($2i$) and check if it is a 3, if so, reject the word, otherwise (or in the case $(i + i)$ exceeds the length of the word) go back left to the first x.

4. Go right until the next 1 is found. While doing it, at each step write on the same cell but in track 3 a 1. If there is not any other 1, accept the word. In case there is another 1, change it by a y and write a 1 on track 3. At this point, we have marked with a y the 1 we are about to check along with the 1 we already marked with an x. Since the second track remained untouched it still contains i the index of x while on the third track, we have j the index of y in unary notation.

5. On the fourth track compute and store the sum of the two indices $(i + j)$.

6. Go to the position $i + j$ on the first track and check its value. If it is a 3 reject the word, otherwise (or in the case $(i + j)$ exceeds the length of the tape) go back to the first y found to the left.

7. Look for the next 1, if it is found change it by a y and go to **step 5**. If there is not any other 1 on the tape change all y on the tape back to 1, position the head of the tape on the rightmost x and go to **step 3**. While going left to find the rightmost x it is important to clear all 1s up to that point in order to have the index of that x on both tracks.

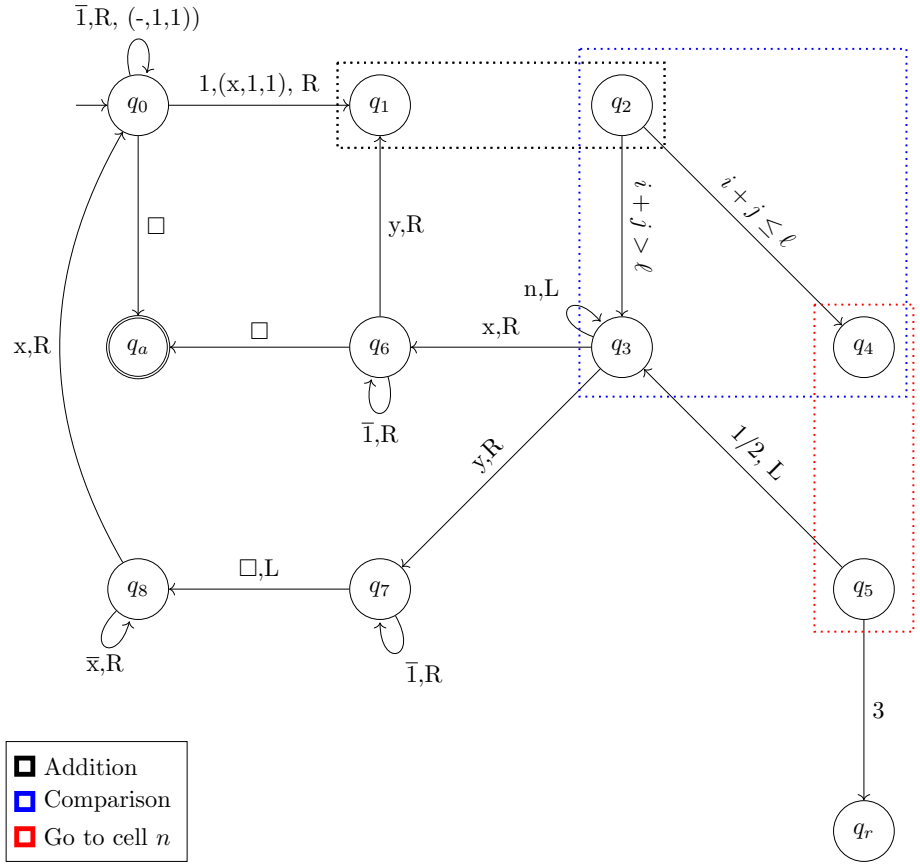


Figure 3.4: Iba that accepts K_3.

In Figure 3.4, the black square represents the subroutine of adding two numbers in unary notation. This corresponds to the input in the second and third tracks. So we shall imagine that when the machine enters state q_1 it computes the addition and returns the output in track four halting in the state q_2. At this point we call a second subroutine, the comparison of numbers. This subroutine is marked with the blue square and contains three states: q_2, q_3 and q_4. After computing the addition of indices this comparing subroutine

checks whether the result is greater than l (the length of the word w) and halts in either q_3 or q_4. Finally, in red, we represent the subroutine of going to a specific position in the word indexed by a natural number in unary notation. This corresponds to the addition of two indices $i + j$ in which there are 1s in the case that the sum does not exceed l.

Remark 3.18. In Figure 3.4, $(-, 1, 1)$ means leave the first track untouched and write a 1 in the second and third tracks, and similarly, $(x, 1, 1)$ stands for the same but writing an x on the first track. Then, a bar over a symbol such as $\overline{x}$ means anything but that symbol, in this particular case anything except for an x. Then, in the case of state q_3, while moving left, it is not written on the graph, but the second and third tracks should be cleared as it is specified in step 7. □

It is clear that if any word contains a 0 or does not contain any 3 it will be rejected. Let us see a couple of examples starting from **step 2** that will make the functioning of the lba clearer.

Example 3.1. $w = 1211213$

As we have already completed **steps 0** and **step 1** we start on the left of the tape with everything blank except for the input and the last track in which we store the length of the word in unary notation. The position at which the machine has its head is represented by a symbol in bold in the first track.

■	1	2	1	1	2	1	3	□	← ***track 1:*** input word w
□	□	□	□	□	□	□	□	□	← ***track 2:*** value index i
□	□	□	□	□	□	□	□	□	← ***track 3:*** value index j
□	□	□	□	□	□	□	□	□	← ***track 4:*** computations
□	1	1	1	1	1	1	1	□	← ***track 5:*** stores the length of the word $\|w\|$

Next, in **step 2** we find the first 1 and cross it and while looking for it, the second and third tracks will get filled with its index in unary notation.

□	**x**	2	1	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	□	□	□	□	□	□	□
□	□	□	□	□	□	□	□	□
□	1	1	1	1	1	1	1	□

The following step, **step 3** consists of three tasks. The first one is to compute the sum of the values in the second and third tracks, that is, to obtain $2i$ being i the index of the first 1 in the word.

□	**x**	2	1	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	□	□	□	□	□	□	□
1	1	□	□	□	□	□	□	□
□	1	1	1	1	1	1	1	□

Still in **step 3**, after having computed $2i$ in the fourth track it is time for theb second task and go to the letter indexed by the value $2i$ and check whether or not it is a 3.

□	x	**2**	1	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	□	□	□	□	□	□	□
□	1	1	□	□	□	□	□	□
□	1	1	1	1	1	1	1	□

Finally, since in position $2i$, in this case 2, there is not a 3, the word is not rejected and we will have to go back to the first x on the left. Having thus completed the last task in **step 3**.

□	**x**	2	1	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	□	□	□	□	□	□	□
□	1	1	□	□	□	□	□	□
□	1	1	1	1	1	1	1	□

After having found the x, now we execute **step 4** and look for the next 1 in the word. While doing it, we record its index by adding 1s to the third track.

□	x	2	**y**	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	□	□	□	□	□
□	1	1	□	□	□	□	□	□
□	1	1	1	1	1	1	1	□

Next, in **step 5** we will just compute the sum of the indices i and j stored in tracks 2 and 3. This process takes place in the fourth track.

□	x	2	**y**	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	1	1	□

In the following step, **step 6** we will do something similar to what has been done in step 3. That is, this step consists of several tasks which are going to index $i + j$ and check weather or not it is a three. In case it was a three we would reject the word, otherwise we go back to the first y on the left (instead of x as in step 3). The next state shows the tape after going to index $i + j$.

□	x	2	y	**1**	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	□	□	□	□	□
□	1	1	1	1	1	□	□	□
□	1	1	1	1	1	1	1	□

Since position $i + j$, that is, position 4 happens to be a 1 we do not reject the word and go back to the first y we find on the left.

□	x	2	**y**	1	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	1	1	□

After having found the y, it is time now to apply **step 7** and look for the next 1 just like we did in step 4.

□	x	2	y	**y**	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	1	1	□

Once we have found the 1 and crossed it by changing it into a y we execute again **step 5**. In other words, we compute $i + j$, the next index to be inspected.

□	x	2	y	**y**	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	□	□	□
□	1	1	1	1	1	1	1	□

At this point, it is time to perform **step 6** and go to position $i + j$, i.e., position 5.

□	x	2	y	y	**2**	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	□	□	□
□	1	1	1	1	1	1	1	□

As in position 5 there is a 2, we do not reject the word and go back to the first y on our left.

□	x	2	y	**y**	2	1	3	□
□	1	□	□	□	□	□	□	□
□	1	1	1	1	□	□	□	□
□	1	1	1	1	1	□	□	□
□	1	1	1	1	1	1	1	□

Now, in **step 7** we look for another 1 on the tape while getting its index on the third track

☐	x	2	y	y	2	**y**	3	☐
☐	1	☐	☐	☐	☐	☐	☐	☐
☐	1	1	1	1	1	1	☐	☐
☐	1	1	1	1	1	☐	☐	☐
☐	1	1	1	1	1	1	1	☐

In this moment, we execute **step 5** to compute $i + j$

☐	x	2	y	y	2	**y**	3	☐
☐	1	☐	☐	☐	☐	☐	☐	☐
☐	1	1	1	1	1	1	☐	☐
☐	1	1	1	1	1	1	1	1
☐	1	1	1	1	1	1	1	☐

Next, in **step 6** we check what happens in position $i + j$, the seventh position.

☐	x	2	y	y	2	y	**3**	☐
☐	1	☐	☐	☐	☐	☐	☐	☐
☐	1	1	1	1	1	1	☐	☐
☐	1	1	1	1	1	1	1	1
☐	1	1	1	1	1	1	1	☐

At this point, we reject the word because we found a 3 in position $i + j$. That is, we are breaking the first Kunz condition ($w_i + w_j \geq w_{i+j}$) for $i = 1$ and $j = 6$. If we had not a 3, for example, if the word was $w = 12112123$, now it would be time to look for another 1 on the tape. Since there would not be any, we would change all y back to 1s, and begin the same process from the x.

We have already seen that K_3 is context-sensitive, now it is time to show that this is the same case for K_q with $q \geq 4$. To do so, we will do the same thing as before, give a high-level description of an lba, but in this case a general one that works for any $q \geq 3$. Since this machine is way more complicated than the previous one, to simplify things we will split the work into some machines, each one dedicated to one job taking advantage of the following fact.

Proposition 3.19. *Context-sensitive languages are closed under intersection.*

Let us go back now to our problem, given a word $w \in \{0, 1, 2 \ldots, q\}^*$, we have that $w = w_1 \cdots w_n$ is an element of K_q if for every pair of indices i, j for which it makes sense, it satisfies the following 4 conditions:

i) $0 \notin w$;

ii) $q \in w$;

iii) $w_i + w_j \geq w_{i+j}$;

iv) $w_i + w_j - 1 \geq w_{i+j-n+1}$.

Since from Proposition 3.19 we know that the intersection of context-sensitive languages is context sensitive, if for every condition above we build an lba capable of recognizing words in $\{0, 1, 2, \ldots, q\}^*$ satisfying that condition, we will have proven that there exists an lba capable of recognizing K_q as a total.
It is easy to see that an lba accepting words with the first two conditions exists. In fact, the language $L = \{w \in \{0, 1, 2 \ldots, q\}^* | 0 \notin w, n \in w\}$ is regular. Next, we will give a high-level description of lbas for each of the last two conditions, the Kunz conditions.

Proposition 3.20. *For every $q \geq 3$ the Kunz language of depth q is context-sensitive.*

Proof. We will describe lbas that are assumed to be able to go to a specific position in the tape indexed by a number, to add and subtract whole numbers and to compare whole numbers. They are designed to have ten tracks, the first track is for the input, and the second track has a copy of the input that will always remain unchanged. The third and fourth tracks are used to write indices of i and j in unary notation. This corresponds to the indices of w_i and w_j that are being checked to verify the Kunz condition, the fifth track is used to compute and store the sum of $i + j$. Similarly, tracks six and seven are used to store in unary notation the values of w_i and w_j and track number eight is used to compute and store their sum. Finally, the ninth track is used to store the length l of the input w and the tenth is used to compare. This tenth track will be used to compare if $w_i + w_j \geq w_{i+j}$ and to compare if $i + j \geq l$, one at a time. Next we can see a table that summarizes each track of the lba and its function.

Track	*Function*	Track	*Function*
1	working copy of w	**6**	store the value of w_i
2	copy of w that remains untouched	***7***	store the value of w_j
3	store index i	***8***	sum $w_i + w_j$
4	store index j	***9***	store |w|
5	sum $i + j$	***10***	compare numbers

Table 3.1: Table showing the function of each track in the design of lbas that recognize K_n.

lba first Kunz condition.

The idea for this lba is to start from the left of the word on index $i = 1$, check if $w_1 + w_1 \geq w_2$, then for every $j > 1$ check if the condition $w_i + w_j \geq w_{i+j}$ holds and then repeat the process for every other $i > 1$. If at any point the condition does not hold, the word will be rejected otherwise, it will be accepted. The letter w_i that is being checked at each time is crossed by changing it into an x, while the other w_j are crossed by changing them into a y. This way we can keep track of which numbers are being checked and it allows us to position the head of the machine. Once all letters are changed to a y it means that the value w_i that was crossed with an x has been verified, then using the fourth track we change every y back to its value and cross with an x the letter immediately right of the previous x and go on with the same process as before. If we end up with all letters crossed with an x the word is accepted.

1. Go right until the first number w_i is found, and at every step right, write a 1 in the same cell but in the third and fourth tracks. Then cross the number by changing it into an x and write its value in tracks six and seven.($track6 = track7 = w_i$). If no number is found and the right end is met, accept the word.

2. Compute and store in track five the sum of the values in the third and fourth tracks. ($track5 = track3 + track4$). If this number exceeds n go to **step 5**.

3. In track number eight compute and store the sum of the values in the sixth and seventh tracks. ($track8 = track6 + track7$)

4. Go to the position indexed by the fifth track ($track5$) and compare its value α with the value in track number eight ($track8$). If ($track8 < \alpha$) reject the word, otherwise move left to the first x.

5. Go right until the first number w_j is found, at every step right, write a 1 in the same cell but in the fourth track. Then write its value on the seventh track ($track7 = w_j$), cross w_j by changing it into a y. Go to **step 2**. If no number is found and the right end is met, change every letter y back to its value and go left until the first x is found and then go to **step 1**.

lba second Kunz condition.

This lba is pretty much the same as the previous one, the main difference is that it checks the word from right to left instead of from left to right and that in order to check the kunz conditions it has to subtract n and add 1. As it goes from left to right, in order to keep an eye on i and j the indices of w_i and w_j, what we will do is initialize the third and fourth tracks with 1s in the same length as the input w and when going left we will erase a 1 by changing it into a $\square$.

0. Go right until the right end is met and find out the length of the word n.

1. Go left until the first number w_i is found, at every step except the last one, change in third and fourth tracks each 1 into a blank ($\square$). Once the number is found, write its value on the sixth and seventh tracks ($track6 = track7 = w_i$), then cross it by changing it into an x. If no number is found and the left end is met, accept the word.

2. Compute and store in the fifth track the sum of the values in the third and fourth tracks ($track5 = track3 + track4 = i + j$), then subtract n and finally add one. That is, $track5 = track3 + track4 - n + 1$. If this number exceeds n or is less than one go to **step 5**.

3. Compute the sum of the values in the sixth and seventh tracks, subtract one and write them in track eight. ($track8 = track6 + track7 - 1$)

4. Go to the position indexed by the fifth track and compare its value α with the value in track eight. If $track8 < \alpha$ reject the word, otherwise move right to the first x.

5. Go left until the first number w_j is found, and at every step except the last one, change in the fourth track each 1 into a blank ($\square$). Once the number is found, write its value on the seventh track ($track7 = w_j$) and cross it by changing it into a y. Then go to **step 2**. If no number is found and the left end is met, change every letter y

back to its value, and every cell in tracks three and four back to 1s as in the original position, after that go right until the first x is found and then go to **step 1**. □

The high-level description of lbas capable of recognizing Kunz languages of depth $q \geq 3$ essentially aims to implement the following algorithm in an automaton giving as much technicalities and formalism as possible without losing the original idea. In particular, checking whether 0 or q belong to the word corresponds to the first two **if**s and could be easily computed with dfas as we stated before. Then every one of the lbas described corresponds to the **if** and **else** inside the **for** loop.

Data: $w = w_1 \cdots w_n \in \{0, 1, \ldots, q\}^*$
Result: true if w is a Kunz word of depth q and false otherwise
if $0 \in w$ **then**
  return *false*;
if $q \notin w$ **then**
  return *false*;
$pairs = \{(i, j) \in \{1, \ldots, |w|\}^2 \mid i \leq j\}$
for $(i, j) \in pairs$ **do**
  if $i + j \leq |w|$ **then**
    if $w_i + w_j < w_{i+j}$ **then**
      return *false*;
  else
    if $w_i + w_j + 1 < w_{i+j-(|w|+1)}$ **then**
      return *false*;
return *true*

Algorithm 1: Algorithm to test whether a word is Kunz.

3.1 Conclusions

Summing up the results in this chapter, we have seen that K_0, K_1 and K_2 are regular languages while K_q, for $q \geq 3$, are context-sensitive and not regular and neither context-free for $q \geq 5$. That is, we have seen that for depth $q \geq 3$ the Kunz languages are more complex than for depth $0, 1$ and 2 and hence we have formalized our intuition that higher values of depth translate to higher levels of difficulty, at least in terms of languages. We have established the distinction between simple and complex in $q = 3$, and this result goes in the same direction as our intuition built upon Wilf's conjecture or the problem of counting numerical semigroups as we mentioned in the introduction. Digging deeper into Wilf's conjecture we have that for depths $0, 1$ and 2 it is easy to verify, for depth 3 it was

harder to check and for depth, 4 it remains unknown whether the conjecture holds or not. Our results do not distinguish between $q = 3$ and $q = 4$, but in the case of this conjecture, there seems to be a difference. Future work in this subject could be checking if K_3 and K_4 are context-free or to find another hierarchy inside context-sensitive languages that separated Kunz languages.

Appendix A

Details for lbas

In the high-level descriptions of lbas in Chapter 3, we assumed that they were capable of adding and comparing two natural numbers within a finite set and going an arbitrary number of steps left or right of the tape. Let us see now, that this can indeed be computed by this kind of machine.

Let us first describe an lba capable of comparing to numbers $n, m, \in \mathbb{N}$ and get to an accepting final state if $n \geq m$ or to a rejecting final state if $n < m$. As input, it will have n and m written in unary notation separated by a 0, as well of course it will have the end markers $[,]$. That is given $n, m \in \mathbb{N}$ the input will be

$$[\underbrace{11\cdots1}_{n}\,0\,\underbrace{11\cdots1}_{m}]$$

, and so the language to be accepted is $L = \{1^n 0 1^m | n \geq m\}$, with the machine always halting.

The formal description of the machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, \square)$ with

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_a, q_r\}$
- $\Gamma = \{0, 1, x, [,]\}$
- $\Sigma = \{0, 1, [,]\}$
- $F = \{\ q_a, q_r\}$

and the partial function δ is defined as follows:

- $\delta(q0, [) = (q_0, [, R), \delta(q0, 1) = (q_0, x, R), \delta(q0, x) = (q_0, x, R).\delta(q_0, 0) = (q_4, 0, R)$
- $\delta(q_1, 1) = (q_1, 1, R), \delta(q_1, 0) = (q_2, 0, R)$
- $\delta(q_2, 1) = (q_3, L, x), \delta(q_2, x) = (q_2, x, R), \delta(q_2,]) = (q_a,], L)$

- $\delta(q_3, 1) = (q_3, 1, L), \delta(q_3, x) = (q_3, x, L), \delta(q_3, 0) = (q_3, 0, L), \delta(q_3, [) = (q_0, [, R)$
- $\delta(q_4, x) = (q_4, x, R), \delta(q_4,]) = (q_a,], L), \delta(q_4, 1) = (q_r, 1, R)$

Next, in Figure A.1 we can see a transition graph representing the lba M. The idea behind its functioning is to cross at each time a 1 from both sides of the 0 and finally detect which side ends up with no number 1 left first.

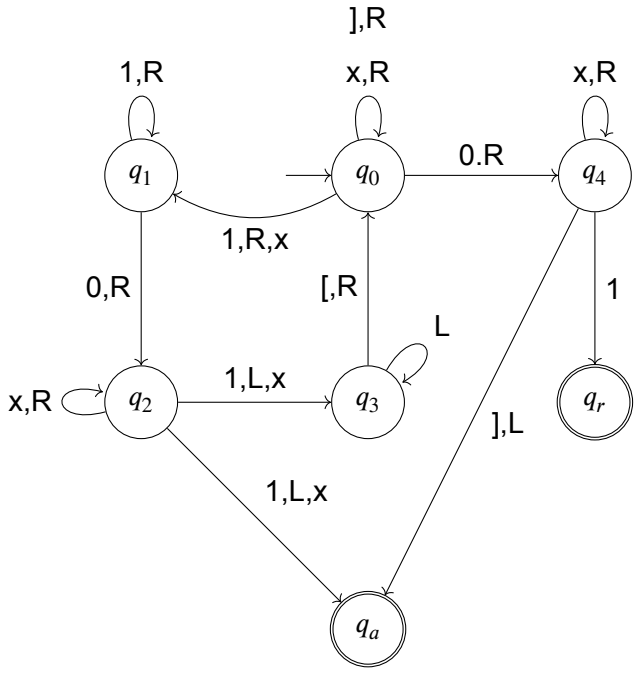


Figure A.1: lba that compares two natural numbers n and m.

We have already shown that an lba can compare natural numbers. Now, we will show that it is able to add numbers as well. In [7, p. 235] there is an example (Example 9.9) that shows with all detail a Turing machine designed for this task. The way it works is very similar to the lba built to compare numbers. In order to add two natural numbers $n, m \in \mathbb{N}$ the automaton writes both of them in unary notation separated by a zero, then the computation of the sum simply consists of moving the zero until the end of the word. That is, the computation of $n + m$ would be:

$$\underbrace{11\cdots1}_{n}0\underbrace{11\cdots1}_{m} \rightarrow \underbrace{11\cdots1}_{n}\underbrace{11\cdots1}_{m}0 = \underbrace{11\cdots1}_{n+m}0$$

Note that this Turing machine designed to add natural numbers is an lba, since the amount of tape needed is the size of the input. If we required a machine to be able to do further computations with the results, such as adding a number to itself repeatedly any

amount of times then the automaton should have an infinite tape.

For both operations, we have used unary notation for natural numbers, but when dealing with Kunz languages the notation used is different. In the definition of Kunz languages, each number $n \in \mathbb{N}_0$ is represented by its own symbol. It is base $n+1$ but without ever exceeding n. Next, we will show that an lba can easily translate back and forth from base $n+1$ to unary notation by giving the transition graph of two Turing machines designed for this task.

To convert a number $m \geq n$ in base $n+1$ to unary, we can use the following machine which gets as input $\square mA \underbrace{\square \cdots \square}_{n}$ and outputs $\square 0A \underbrace{11 \cdots 1}_{m} \underbrace{\square \cdots}_{n-m}$.

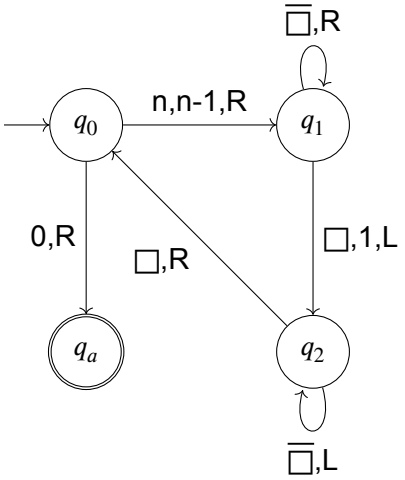


Figure A.2: Turing Machine that converts a natural number from base $n+1$ to unary notation.

Note that the previous Turing machine when fixed the number n is an lba, since the amount of tape is limited to $n+2$ cells no matter what.

Now, for the converse, translating unary to base $n+1$ can be done with the following machine. which gets as input $m \geq n$ written as $[\underbrace{11 \cdots 1}_{m} A0]$ and outputs $[\underbrace{00 \cdots 0}_{m} Am]$

.

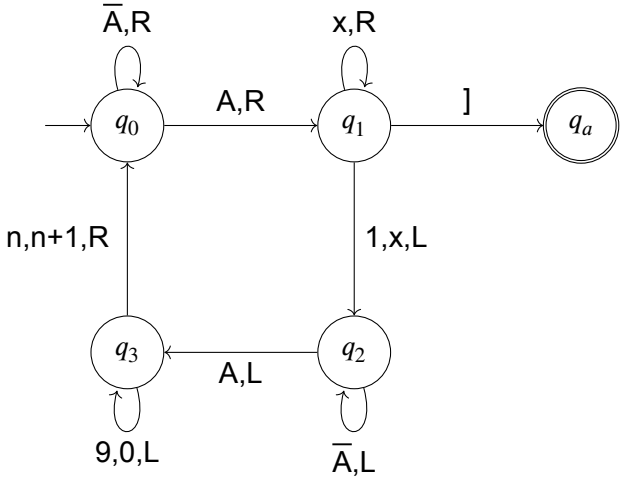


Figure A.3: lba that converts a natural number from unary to base $n+1$ notation.

Once more this Turing machine is clearly an lba, so with this, we have already made it clear that we can ask our previous machines to add and compare numbers without any problem.

Finally, to position the head of the machine in some index $i \in \{1, \cdots, |w|\}$ in the tape is quite easy. Note that i will be written in some track in unary notation ,so that track along with the track with the input word will look like the orders are as simple as going left until

$\square$	w_1	w_2	$\cdots$	w_i	w_{i+1}	$\cdots$	$w_{\lvert w\rvert}$	$\square$	$\cdots$	$\square$
$\square$	1	1	$\cdots$	1	$\square$	$\square$	$\cdots$	$\square$	$\cdots$	$\square$

the beginning and then go forward until the last one. This automaton can be represented as:

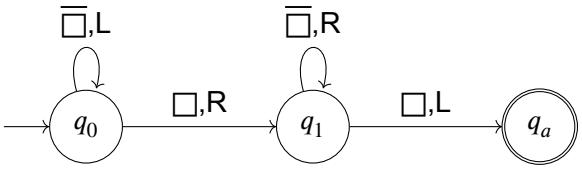


Figure A.4: Turing Machine that goes to index $i \in \{1, \ldots, |w|\}$.

Subtraction, actually *proper subtraction* can be done using a TM as the one specified in Example 8.4 [5, p. 332].

Bibliography

[1] Christopher Bader and Arnaldo Moura. “A Generalization of Ogden’s Lemma”. In: *J. ACM* 29.2 (Apr. 1982), pp. 404–407. url: https://doi.org/10.1145/322307.322315.

[2] Manuel Delgado. “Conjecture of Wilf: A Survey”. In: *Numerical Semigroups*. Ed. by V. Barucci et al. Vol. 40. Springer INdAM Ser. Springer, Cham, 2020, pp. 39–62.

[3] Manuel Delgado and Jaume Usó i Cubertorer. *Kunz languages for numerical semigroups are context sensitive*. 2023. arXiv: 2306.03308 [cs.FL].

[4] Shalom Eliahou and Jean Fromentin. “Gapsets and numerical semigroups”. In: *Journal of Combinatorial Theory, Series A* 169 (Jan. 2020), p. 105129. url: https://doi.org/10.1016%5C%2Fj.jcta.2019.105129.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[6] Nathan Kaplan. “Counting numerical semigroups”. In: *Amer. Math. Monthly* 124.9 (2017), pp. 862–875. url: https://doi.org/10.4169/amer.math.monthly.124.9.862.

[7] Peter Linz. *An Introduction to Formal Languages and Automata, Fifth Edition*. 5th. USA: Jones and Bartlett Publishers, Inc., 2011.

[8] Alberto Pettorossi. *Automata Theory and Formal Languages: Fundamental Notions, Theorems, and Techniques*. Undergraduate Topics in Computer Science. Springer International Publishing, 2022. url: https://doi.org/10.1007/978-3-031-11965-1.

[9] J. L. Ramírez Alfonsín. *The Diophantine Frobenius problem*. Vol. 30. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford, 2005, pp. xvi+243. url: https://doi.org/10.1093/acprof:oso/9780198568209.001.0001.

[10] J. C. Rosales and P. A. García-Sánchez. *Numerical semigroups*. Vol. 20. Developments in Mathematics. Springer, New York, 2009, pp. x+181. url: https://doi.org/10.1007/978-1-4419-0160-6.

[11] J. C. Rosales et al. "Systems of inequalities and numerical semigroups". In: *J. London Math. Soc. (2)* 65.3 (2002), pp. 611–623. url: https://doi.org/10.1112/S0024610701003052.

[12] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013.

[13] Alex Zhai. *Fibonacci-like growth of numerical semigroups of a given genus*. 2011. url: https://arxiv.org/abs/1111.3142.

[14] Yufei Zhao. "Constructing numerical semigroups of a given genus". In: *Semigroup Forum* 80.2 (Oct. 2009), pp. 242–254. url: http://dx.doi.org/10.1007/s00233-009-9190-9.

[15] Daniel G. Zhu. *Sub-Fibonacci behavior in numerical semigroup enumeration*. 2022. url: https://arxiv.org/abs/2202.05755.